软件分析与架构设计

# 软件架构设计

何冬杰
重庆大学

# 软件开发历史

*1970’*
- 汇编语言
- 适用于小型程序
- 高级语言

*1980’*
- 面向过程理论
- 数据流/控制流设计方法
- 应用程序开发库 （类库/函数库）

*1995*
- 面向对象理论
- 面向对象建模以及设计技术
- 应用程序开发框架: J2EE, .NET
- 组件技术: COM/DCOM, CORBA …

**Future**
- 对象建模/设计标准化: UML
- 模型驱动开发: MDA
- …

## 软件演化趋势

- **软件的规模和复杂度越来越高**

  应用领域: 科学计算、工业生产、商业、教育、娱乐

- **软件的抽象程度越来越高**

  机器语言 - >汇编语言 - >高级语言 - > 开发框架面向过程编程 - > 面向对象编程 - > 面向切面编程

# 为什么要做架构设计?

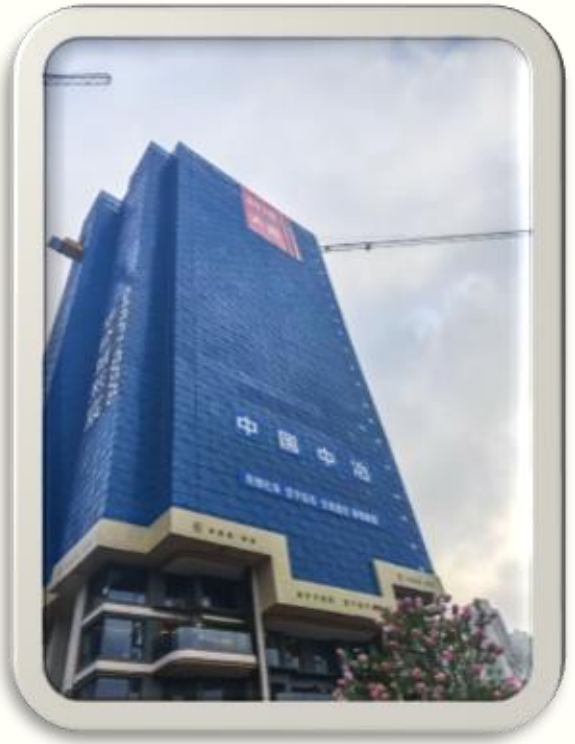
- 规模
- 过程
- 工作计划
- 风险
- 开发团队实力
- 成本
- 所需技术
- 利益相关者

**一人就可完成构建**

- 无需建模
- 过程简单
- 工具简单

**团队设计更为高效**

- 需要建模
- 健全的过程
- 有力的工具



**更加专业的设计团队**

- 安全性验证
- 成本风险评估等

# 软件架构设计

**定义**

Software architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution ----- IEEE 1471-2000

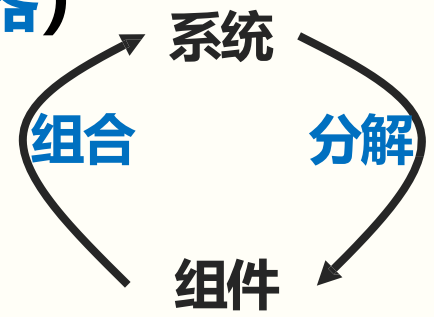
- **软件元素**：功能、接口、程序、类模块、层、子系统、客户端/服务器等
- **可见属性**： 提供服务、性能特征、错误处理、公享资源使用等
- **关系**：这些元素之间的组合机制

## ❑软件架构**是商业和技术决策的结果**

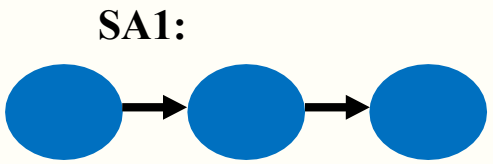
- 好的架构设计对于软件系统成败至关重要
- 架构设计比数据结构和程序算法更为重要

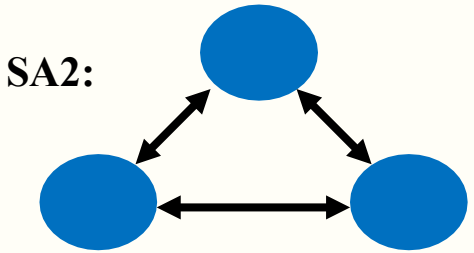
# 软件架构相关概念

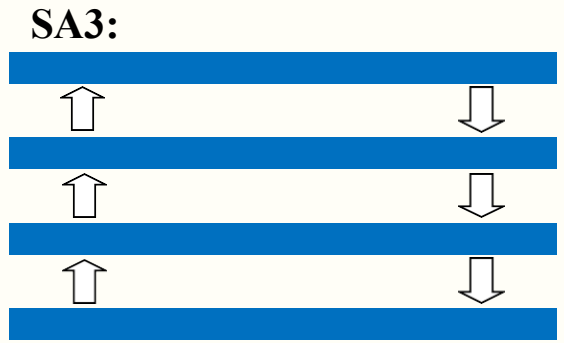
- **软件系统：构成元素（组件）及其交互机制（连接器）**




- **组件：软件系统的逻辑或功能单元**
  - 组件可以被拆分为更小的单元或者组件
  - 组件提供特定的职责
  - 组件是抽象的、概念的词语，在不同的场景中可以是不同的特定对象
    - 例如：模块、子系统、层、包、类等
- **连接器：组件之间的交互规则或机制**



# 软件架构相关概念

## 分解/组合

- 降低软件设计和构建的复杂度
- 控制软件开发的风险
- 提高组织管理的效率

**需要考虑的问题：**

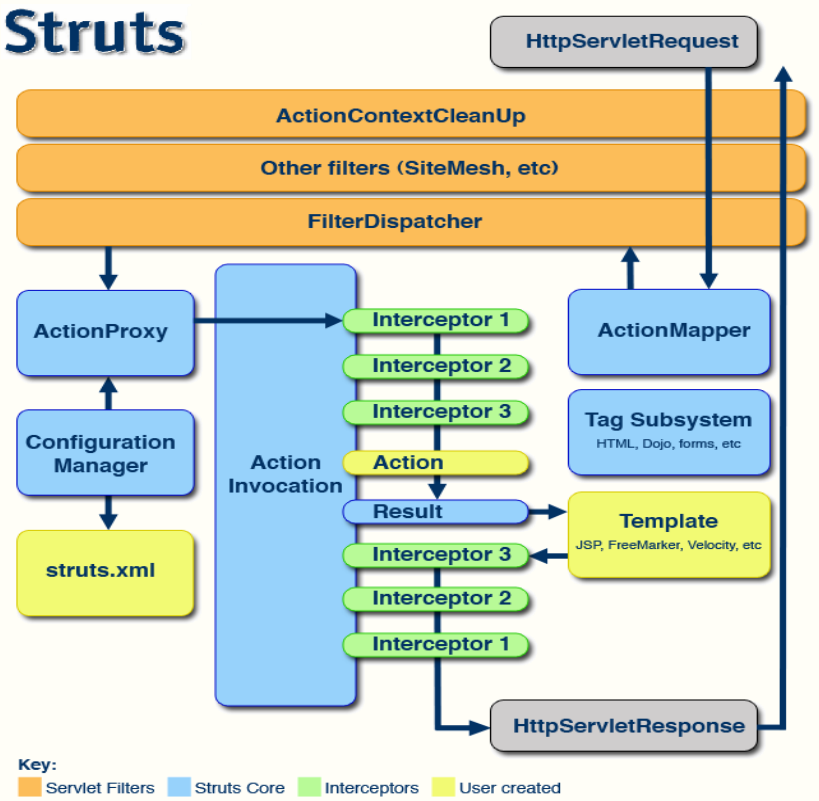
- 如何将系统分解为组件？
- 是否拥有所有必须的组件？
- 组件能否进行集成？

## 功能/非功能属性

- **软件架构的功能属性**
  - 满足功能性需求的属性
- **软件架构的非功能属性**
  - 满足非功能性需求的属性
    - 性能
    - 可移植性
    - 灵活性/可扩展性
    - 可靠性/安全性
    - …

# 软件架构相关概念：框架

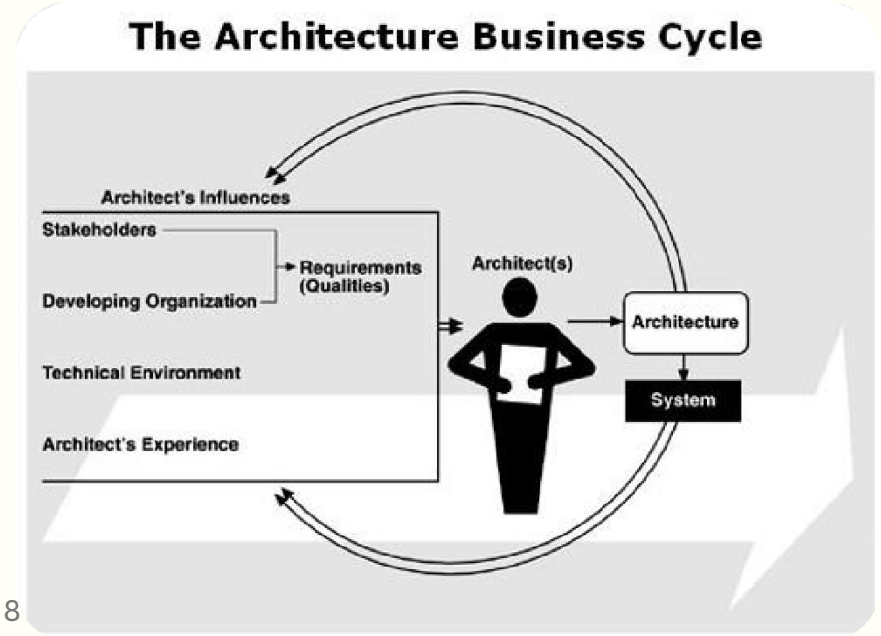
- **框架是解决特定问题的可重用的应用程序结构**
  - 提供解决特定问题的必要的、基本的组件
  - 提供组件间的交互机制和约束
  - 提供基于框架开发的应用程序的上下文或环境
  - 通常而言，框架以类库形式提供，
    - 例如: .NET 框架, JavaEE框架，Spring框架等

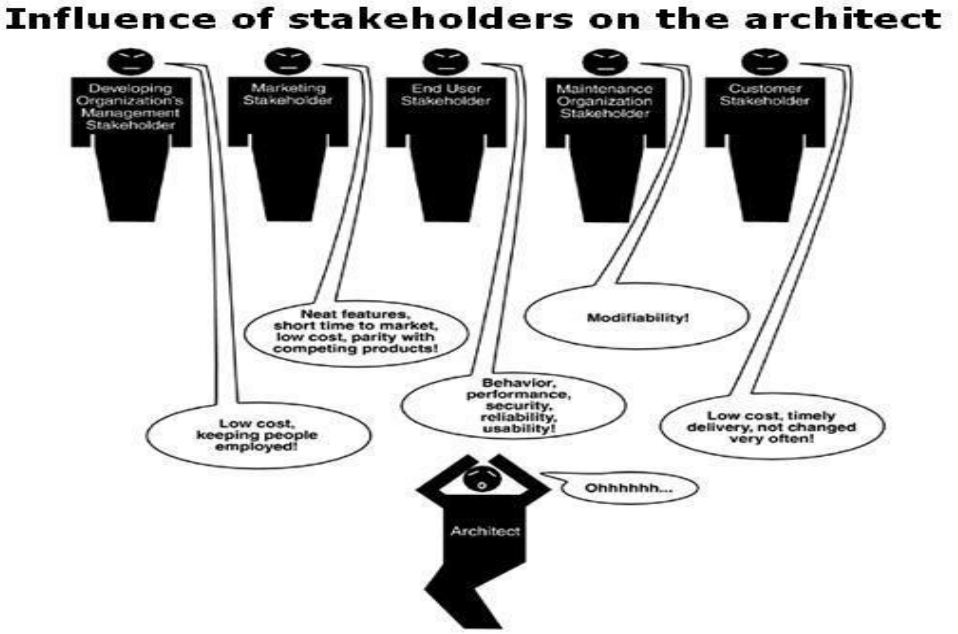


# 软件架构设计的影响因素

- ❑架构受系统利益相关者影响
- ❑架构受开发组织影响
- ❑架构受架构师背景和经验影响
- ❑架构受技术环境影响

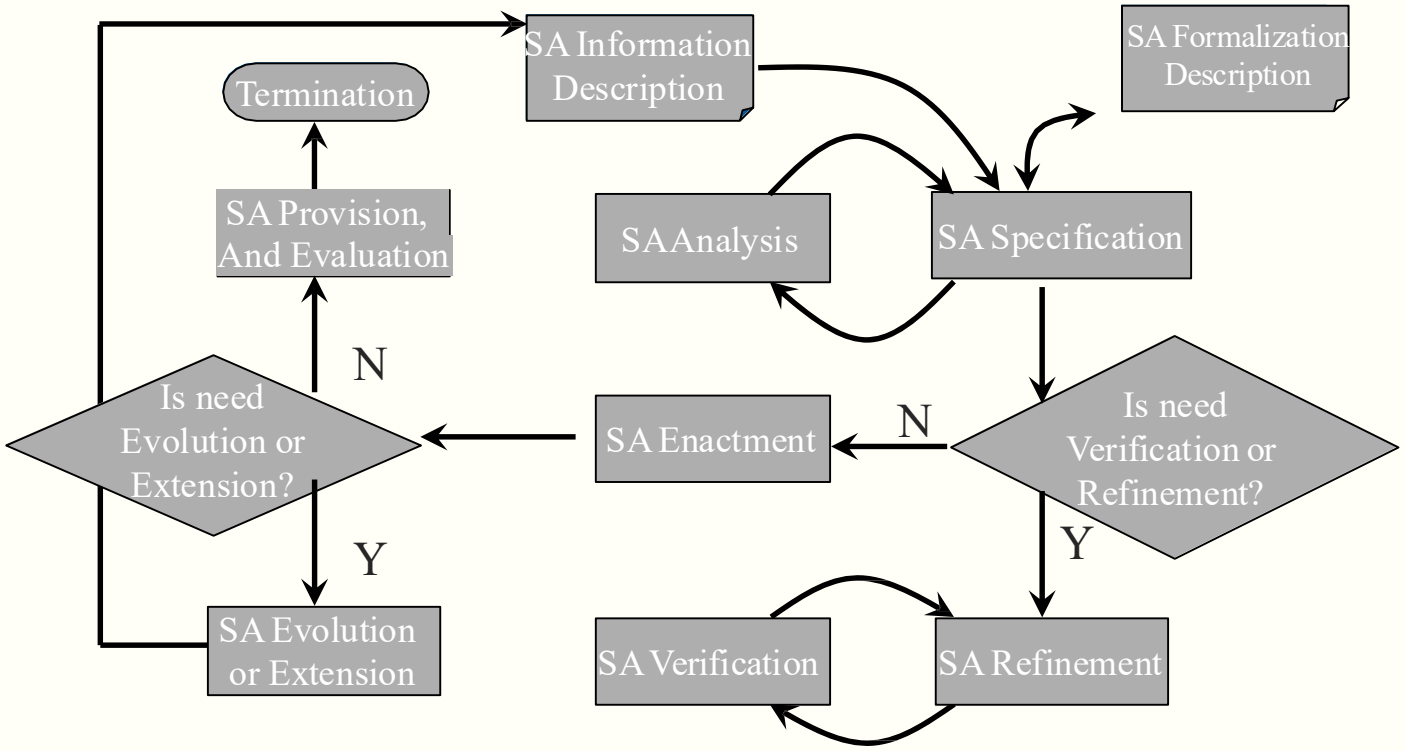
- 用户、客户
- 项目经理
- 架构师、开发人员
- 系统工程师、运营人员
- 其他开发者



# 软件架构设计流程及生命周期模型

- **设计活动流程**
  - 创建系统的业务用例
  - 理解用户需求
  - 创建或选择合适架构
  - 分析评价架构
  - 实现基于架构的系统
  - 实现基于架构的系统
  - 确保系统实现符合架构




生命周期模型

# 软件架构及其价值

- 架构是系统可看为一个整体的高层次的抽象
- 在架构层次，隐藏所有的实现细节。
- 架构必须支持系统的功能性需求
- 架构必须满足系统的非功能性需求: 性能、安全性、可靠性、灵活性、可拓展性等。

## ❑组织层次

- **有助于组织内部，或者客户和开发商间的沟通**
- **提供系统更高层次的抽象**
- **成本和风险评价**
- **工作分配和项目规划**

## ❑技术层面

- **满足系统需求和目标**
- **规定详细设计、构建、测试的约束**
- **确保对系统的灵活划分**
- **减少维护和演化的成本**
- **增加重用，与旧的、第三方软件结合**

# 好架构的特征

- ❑弹性
- ❑简单
- ❑可实现
- ❑功能职责的清晰分解
- ❑平衡职责分布
- ❑平衡经济和技术约束

软件分析与架构设计

# 软件架构建模

何冬杰
重庆大学

# 为什么要进行软件架构建模?

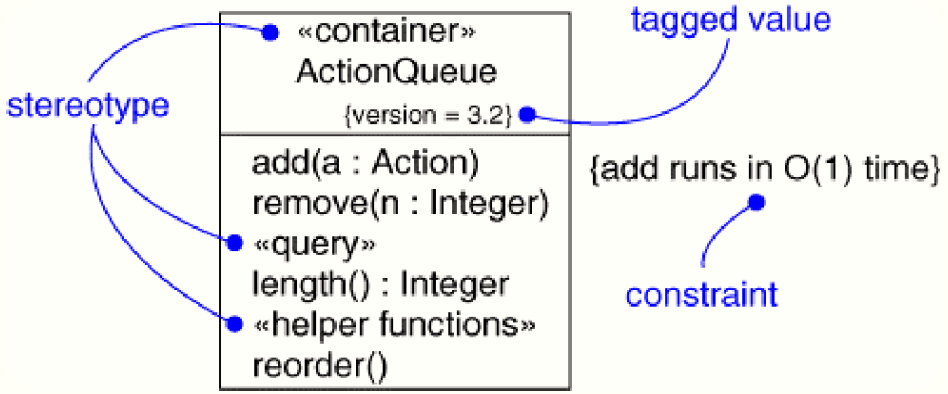
- **模型**：对无法直接观测的对象的可视化描述或近似
  - 模型是对现实的简化，提供了系统的蓝图
  - 模型可以是结构化的，着重描述系统的组织架构
  - 模型也可以是行为化的，着重描述系统的动态行为
- **建模可以让我们更好地理解构建的系统：**
  - 对系统进行可视化
  - 规约系统的结构或行为
  - 用于指导构建系统的模板
  - 将设计决策形成为文档

# 软件架构建模语言和工具：UML

- UML语言适用于对软件密集型系统的物件进行：
  - 可视化
  - 规范化
  - 构建
  - 文档化

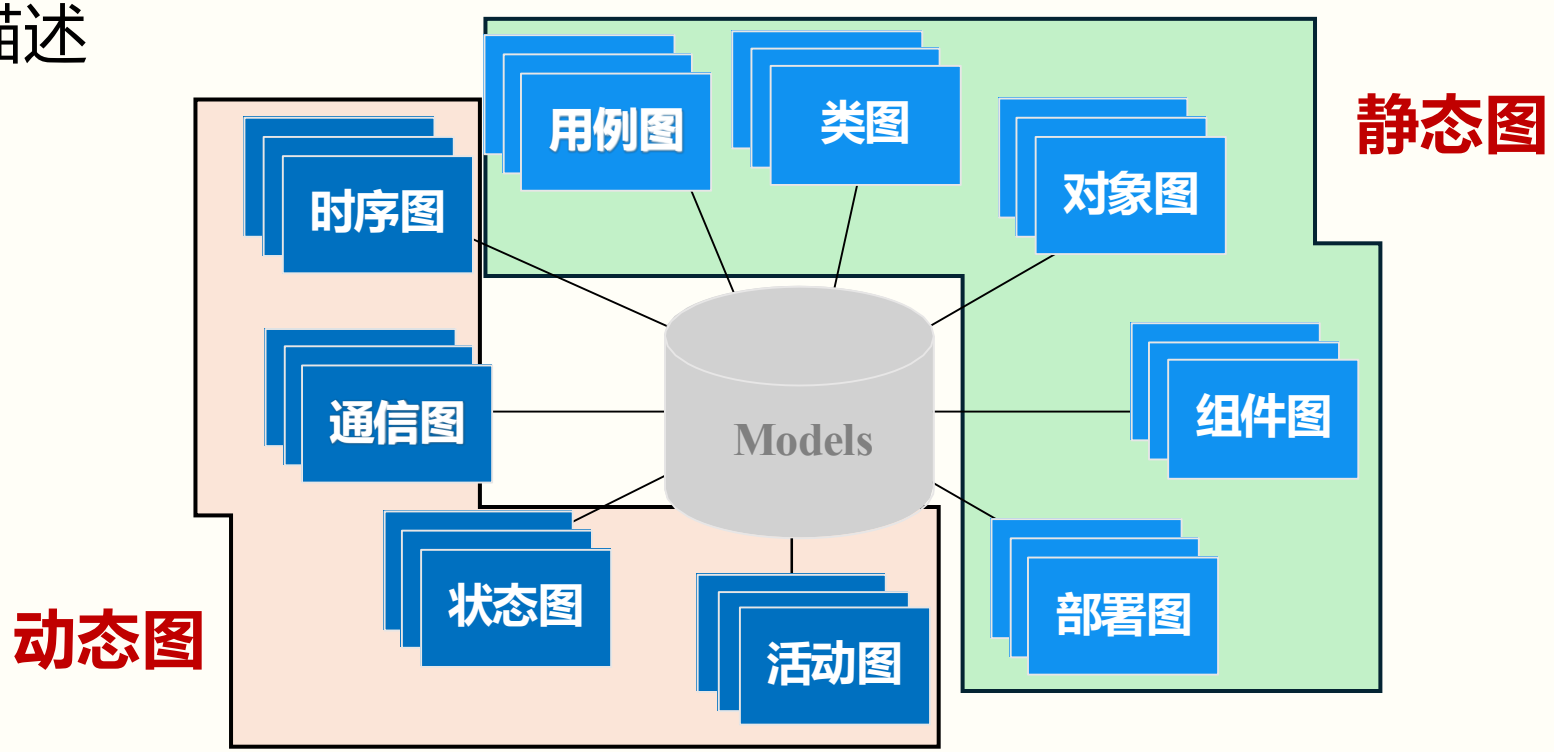

- **相关工具：**
  - **Microsoft Visio、Lucidchart、StarUML、Umbrello、Modelio**

- UML语言：
  - 建模元素： 结构化元素（类、接口、协作、用例组件、节点）；行为元素（交互、状态机）；分组元素（包、子系统）；其他元素（注释）
  - 关系：依赖、关联、泛化、实现
  - 扩展机制：构造型、标记值、约束
  - 图



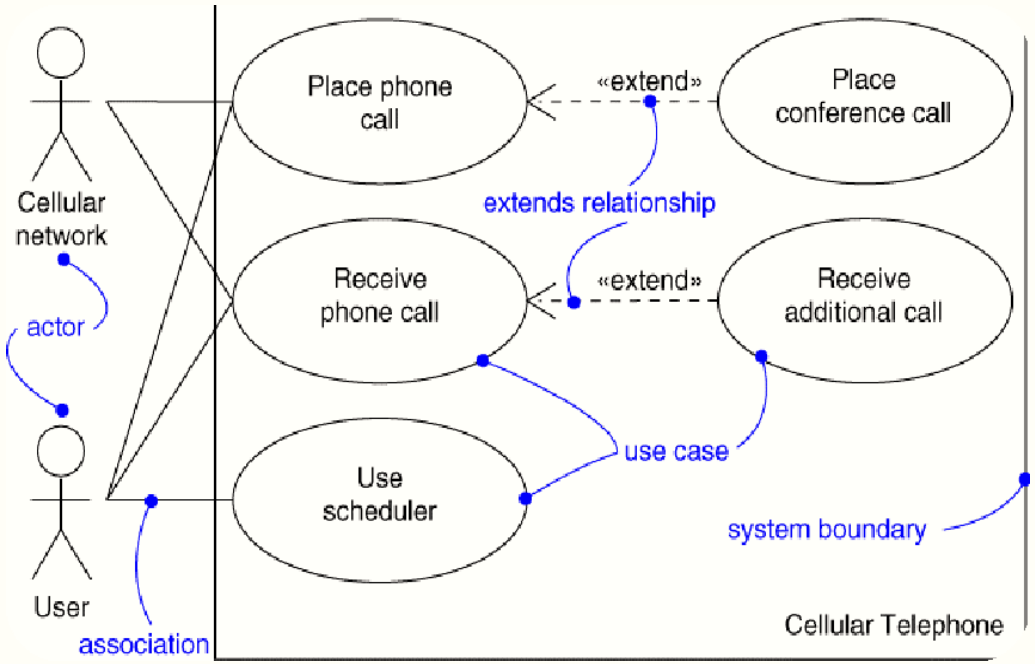
# 模型与视图

- 模型是从多个特定的视角对系统的完整描述
- 图是模型的一个视角
  - 从某个特定的利益相关者的视角展示系统
  - 提供系统的局部描述
- UML视图
  - 静态视图:
  - 动态视图:



# 用例图：从用户视角描述系统功能需求

- **开发早期阶段使用**
- **目的：**
  - 确定系统上下文
  - 捕捉系统需求
  - 验证系统架构
  - 驱动系统实现和测试用例生成
- **由领域专家和系统分析师完成**

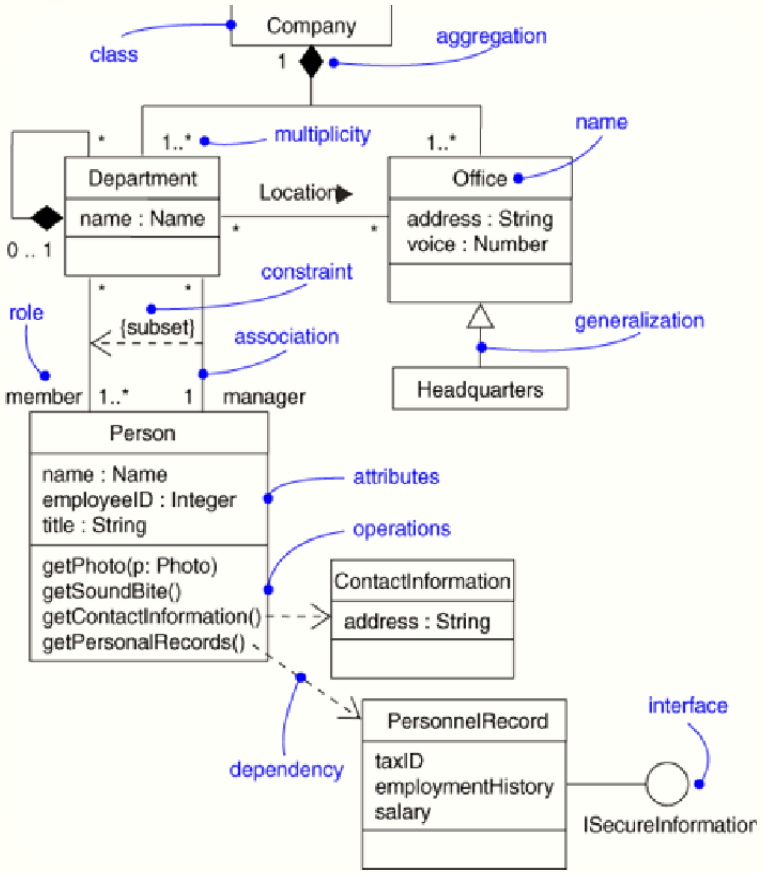


# 类图： 捕捉系统的词汇表

## ❑目的：

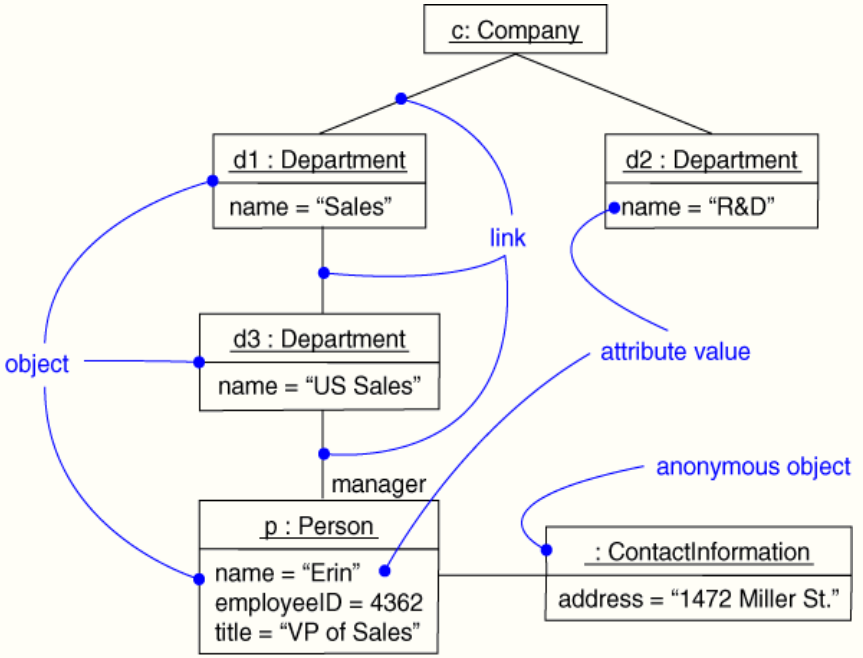
- 对系统中的概念进行命名和建模
- 确定协作关系
- 确定逻辑数据库结构

## ❑由分析师、设计师和开发人员完成



# 对象图：描述对象实例间的关系

- 在分析和设计阶段绘制
- 目的：
  - 描述数据/对象结构
  - 某个时间点的系统快照
- 由分析师、设计师、开发人员共同绘制

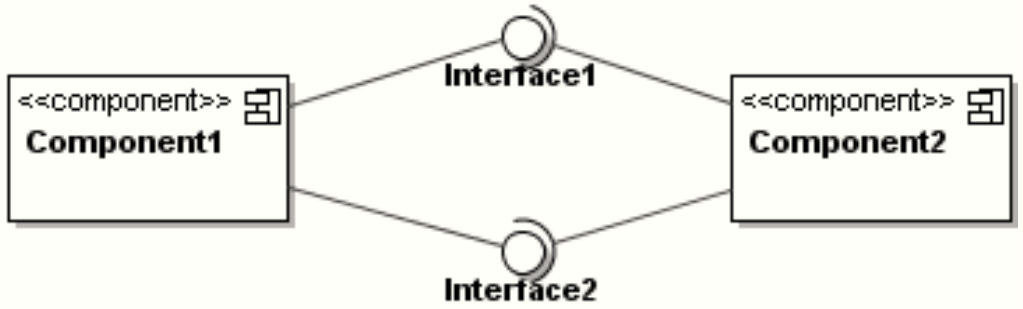


# 组件图与部署图

## ❑组件图：

- ➢描述系统实现的物理结构
- ➢组织源代码
- ➢构建可执行的release版本
- ➢表述物理数据库

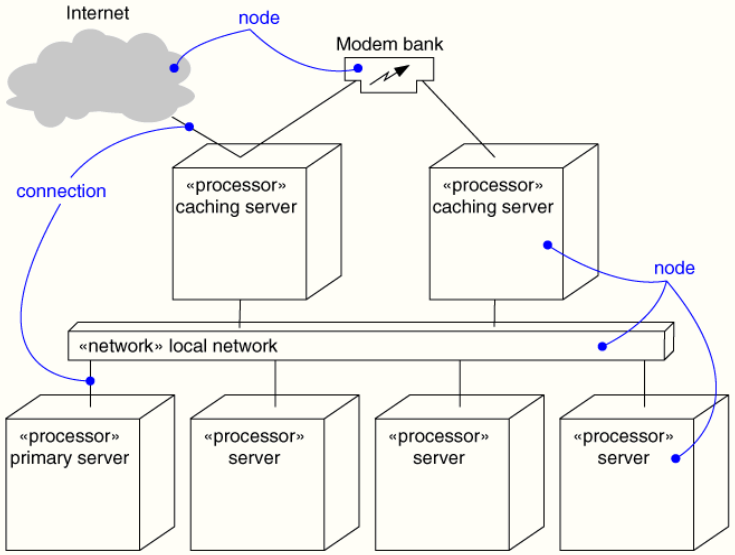
## ❑由架构师和程序员绘制




## ❑部署图：

- ➢描述系统硬件的拓扑结构
- ➢描述组件的硬件部署
- ➢识别性能瓶颈

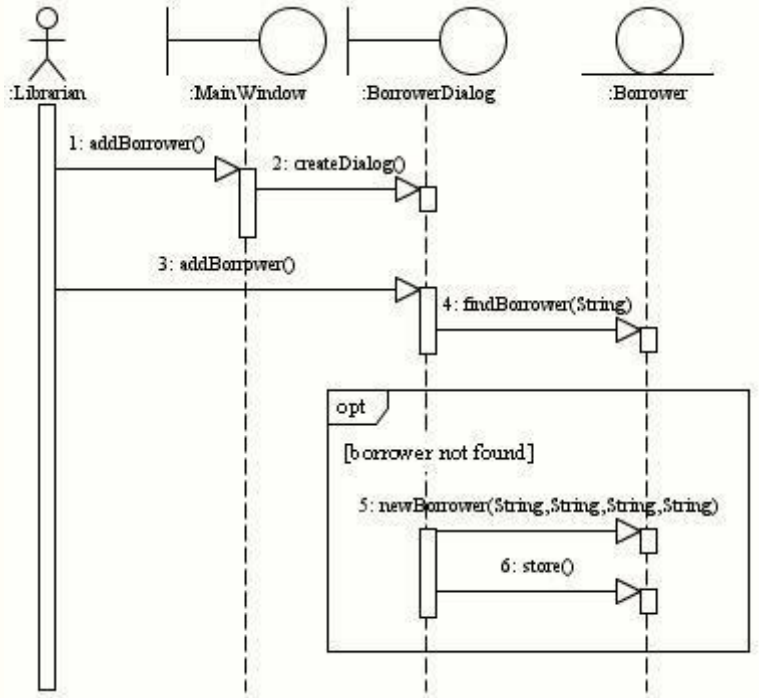
## ❑由架构师、网络工程师、系统工程师共同绘制



# 时序图与通信图

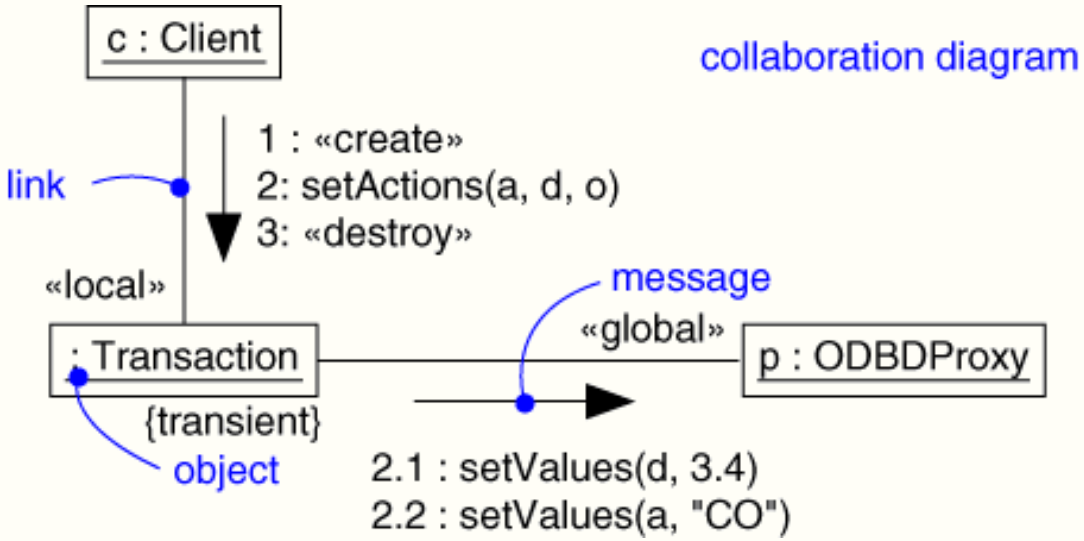
## ❑时序图：

- ➢描述面向时间的动态行为
- ➢对控制流进行建模
- ➢描述用例场景




## ❑通信图：

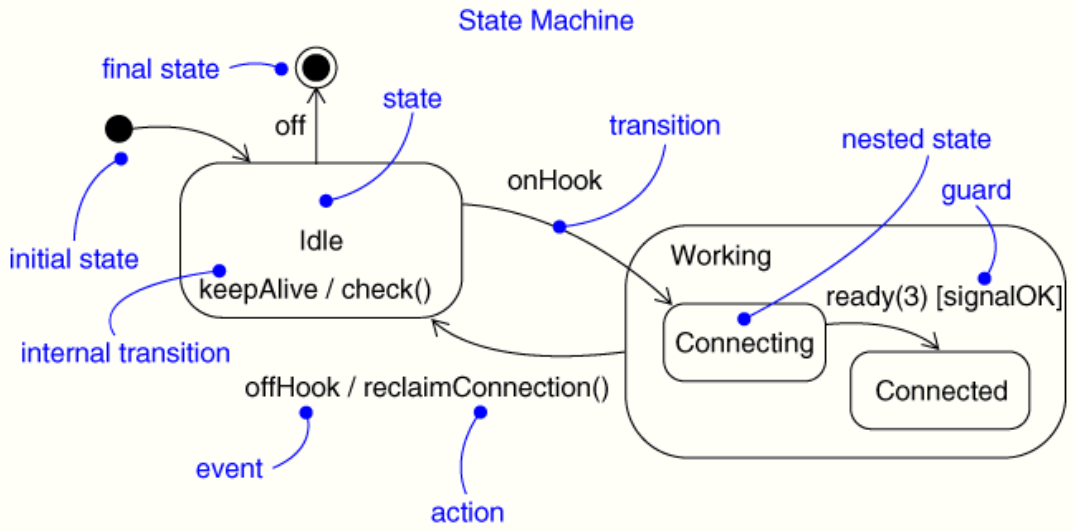
- ➢描述面向消息的动态行为
- ➢对控制流进行建模
- ➢描述对象结构和控制的协调



# 状态图与活动图

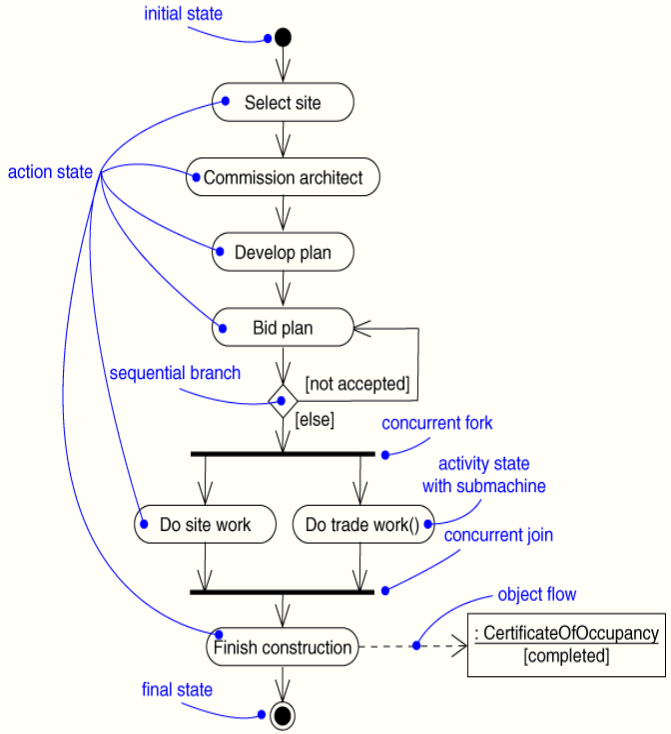
## ❑状态图：

- ➢描述面向事件的动态行为
- ➢为对象生命周期进行建模
- ➢为响应对象建模（用户界面、设备等）




## ❑活动图：

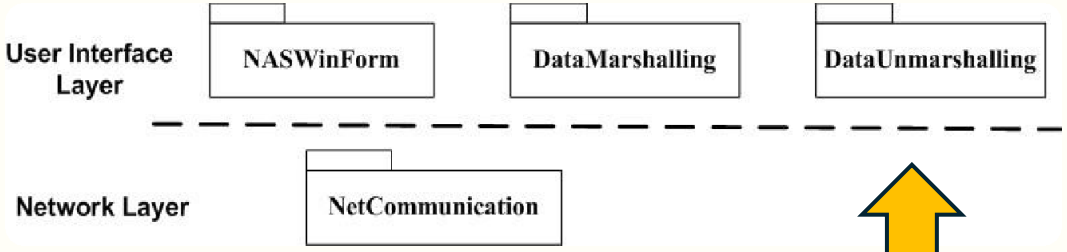
- ➢描述面向活动的动态行为
- ➢为业务工作流建模
- ➢为操作建模



# 架构视图

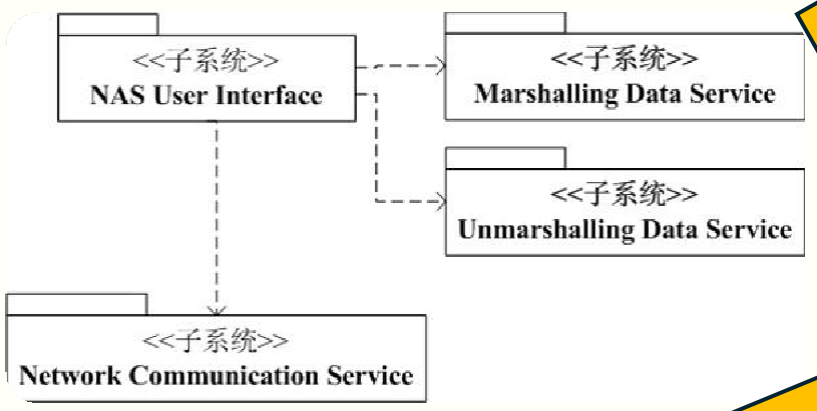
- **定义：从特定的视角对系统的简化描述（抽象），覆盖特定的关注点，忽略那些与该视角无关的实体对象**
- **存在问题：**
  - 架构视图过度强调开发方面，没有解决所有利益相关者关注的问题
  - 软件系统的利益相关者: 用户、开发者、系统工程师、项目经理
  - 软件工程师极力添加更多的信息到架构图中,导致架构图复杂。
- **解决方案：**
  - 采用多个并发视图, 每个视图用不同的符号展示不同的关注点
  - “4+1” 视图模型为大的、复杂的架构建模。

# 4+1视图模型



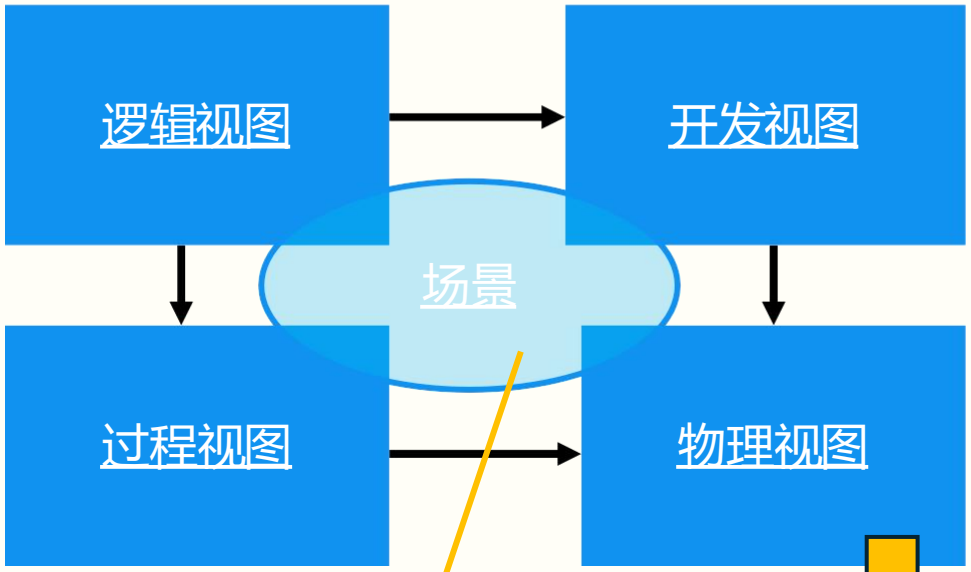

- 系统的功能抽象。将系统分解为多个功能组件，描述其功能关系。

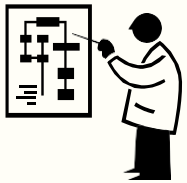
- 系统的详细设计与实现的抽象。




用户

程序员
& 软件管理人员



集成工程师

系统工程师

- 主要描述系统的非功能性需求和系统的运行特征

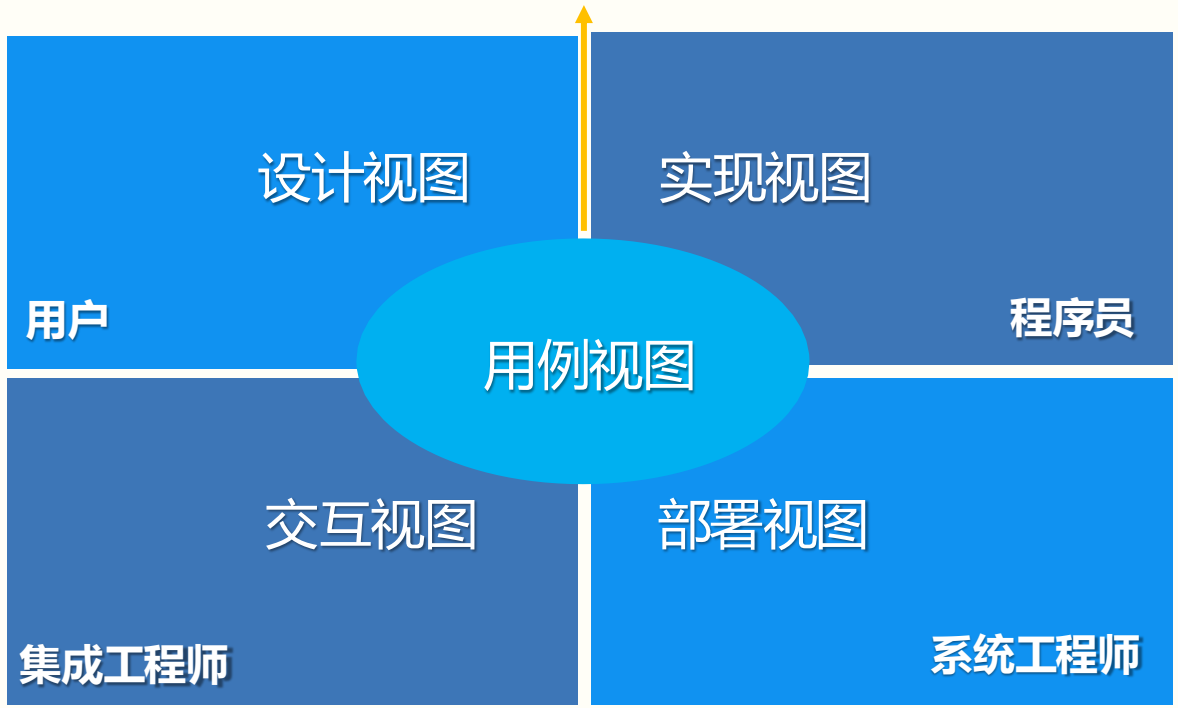
- 描述系统重要的业务用例

- 描述系统的硬件拓扑结构，以及软件组件在硬件上的部署。

# Rational的4+1视图模型

- 包含描述用户、分析师和测试工程师看到的系统行为的用例。根据视图可确定系统架构。

- 包含构建系统的类、接口和类之间的协作；主要支持系统的功能性需求，即系统提供给用户的服务。




- 包含用于组装和发布物理系统的组件；主要解决系统发布的配置管理问题。

- 描述了系统不同部分之间的控制流，包括可能的并发和同步机制；主要解决系统的性能、可拓展性、吞吐量等问题。

- 包含形成系统硬件拓扑结构的节点；主要解决构成物理系统的部件的分布、发布和安装问题。

- 不是所有系统都需要所有视图: 单一处理器: 舍弃部署视图；单一进程: 舍弃交互视图；小程序: 舍弃实现视图
- 也可添加视图: 数据视图、安全视图


# 软件架构设计原则

**何冬杰**

**重庆大学**

# 不良架构设计有哪些特征?

- 刚性、脆弱性、不动性、粘性、不必要的复杂性和重复、不透明度
- 刚性：系统难改变，每次改变都会迫使系统其他部分发生变化
  - 根本原因是模块化不好，耦合度高
    - 耦合描述一个对象如何依赖另一个对象；松散耦合的对象可以根本地改变，而不会相互影响；对紧密耦合的对象稍作修改可能会导致一系列问题。
    - 例：类A和B紧耦合，修改A中的x域，需要同步修改类B

```
class A {
  int x;
  ...
}
```

```
class B extends A {
  void b() {
    x = 5;
  }
}
```

# 不良架构设计有哪些特征?

- 脆弱性：更改会导致系统在与更改无关部分中断或崩溃
- 不动性：将系统难以分解成可在其他系统中重用的组件
- 粘性：做正确的事情比做错事更困难
  - 如果进行保留设计的变更比做"黑客"更难，那么设计的粘度就会很高；
  - 当开发环境缓慢且效率低下时，环境粘度就会产生
- 不必要的复杂性：包含的基础设施不会带来直接的好处
  - 设计将承载所有未使用的设计元素的重量，并可能使其他更改变得困难
- 不必要的重复：包含可在单一抽象下统一的重复结构
  - 剪切和粘贴的结果； 代码几乎相同的半重复
- 不透明度：代码很难阅读和理解;没有很好地表达它的意图

# 架构设计原则

- **设计原则是设计的指导方针，原则的遵循可避免糟糕的设计**
- **常见面向对象（OO）设计原则**
  - SRP - 单一职责原则
  - OCP - 开闭原则
  - LSP - 里氏替换原则
  - ISP - 接口分离原则
  - DIP - 依赖倒置原则

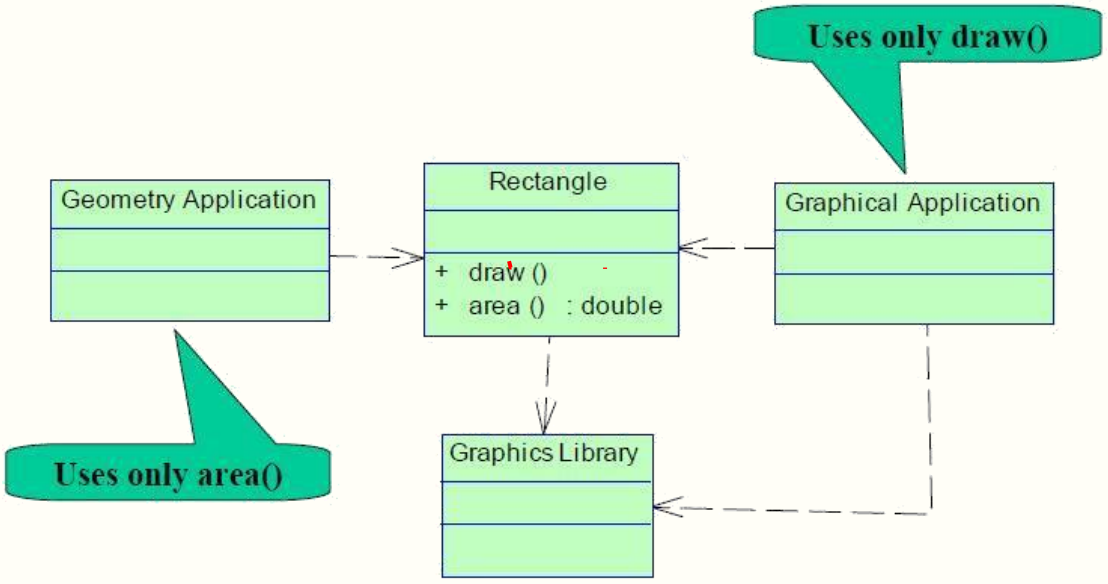
# SRP - 单一职责原则

- **一个类只有一个改变的理由 （职责 =“改变的理由”）**
  - 如果一个类有多个职责，那么职责就会彼此耦合
  - 如果模块具有多个职责，内聚性就低
  - 良好设计的内聚性应该很高
- **内聚性：**
  - 面向对象编程中，内聚性指方法实现单一功能的程度。
  - 实现单一功能的方法具有高内聚性
  - 低内聚性的表现：一个类的方法很少相关；方法执行不相关的活动，通常使用不相关的数据集。
  - 低内聚性的缺点：模块理解困难；系统维护困难；模块重用困难

# SRP - 单一职责原则：例子I

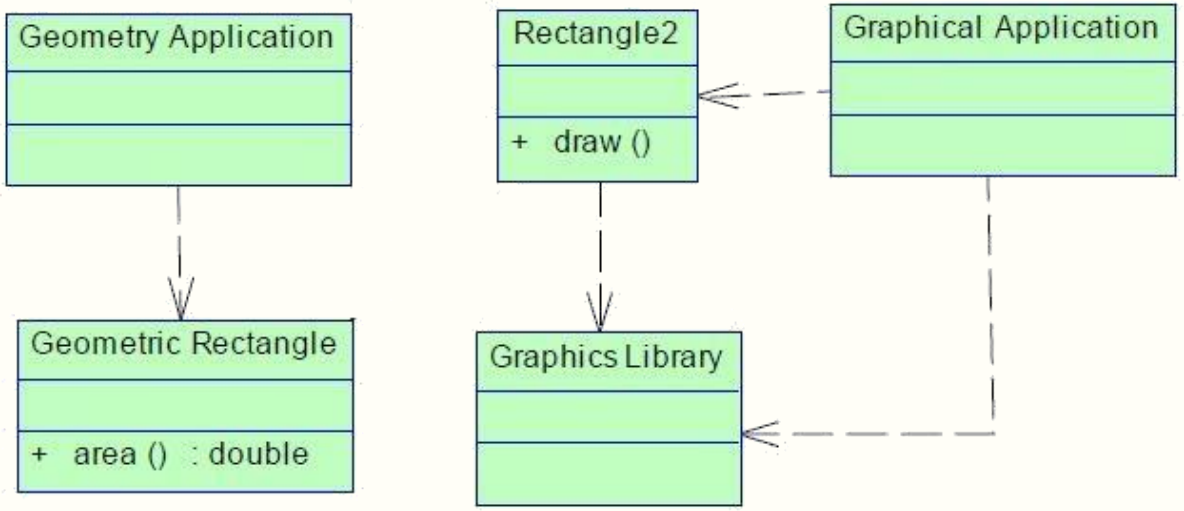
## 不良设计

❑Rectangle类有两个职责




## 更好的设计

❑创建2个类，职责分离




**注意：**如果职责不会独立发生变化，职责无需分离。 否则，会造成"不必要的复杂性"。

# SRP - 单一职责原则：例子II

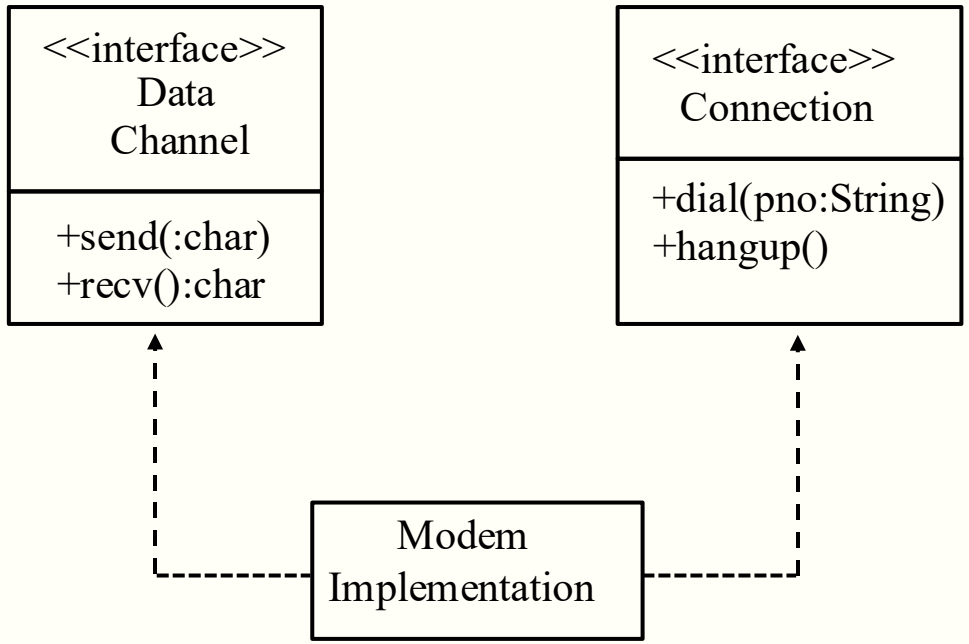
## 不良设计

❑胖接口

```
interface Modem {
  void dial(String pno);
  void handup();
  void send(char c);
  char recv();
}
```

## 更好的设计

❑拆分接口



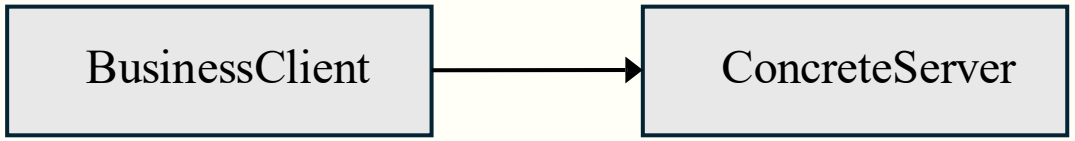
# OCP – 开闭原则

## ❑对扩展是开放的；对更改是封闭的

- ➢这意味着需求发生变化时，以对原有设计进行扩展方式应对，而不是以修改原有代码的形式应对

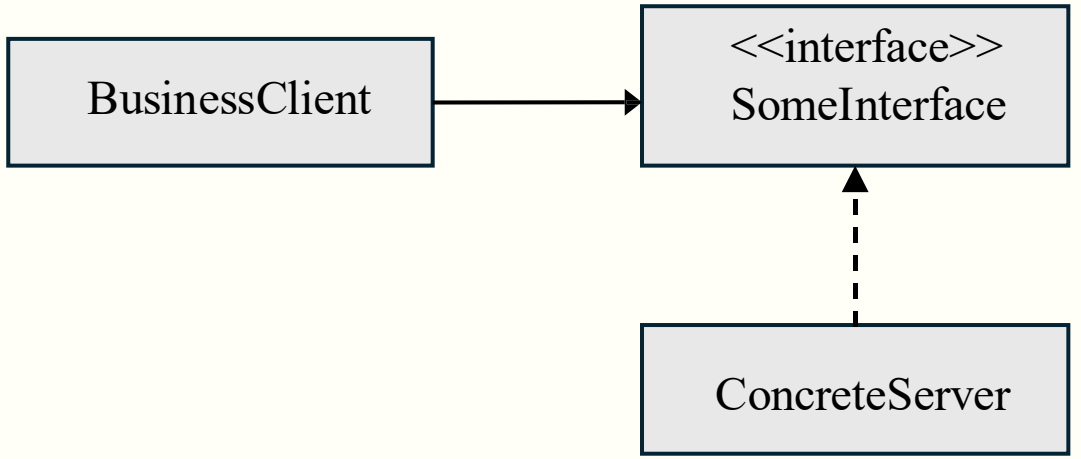
## ❑违反 OCP原则的例子

- ➢如果BusinessClient类具有对ConcreteServer类的引用，则替换ConcreteServer会导致修改BusinessClient

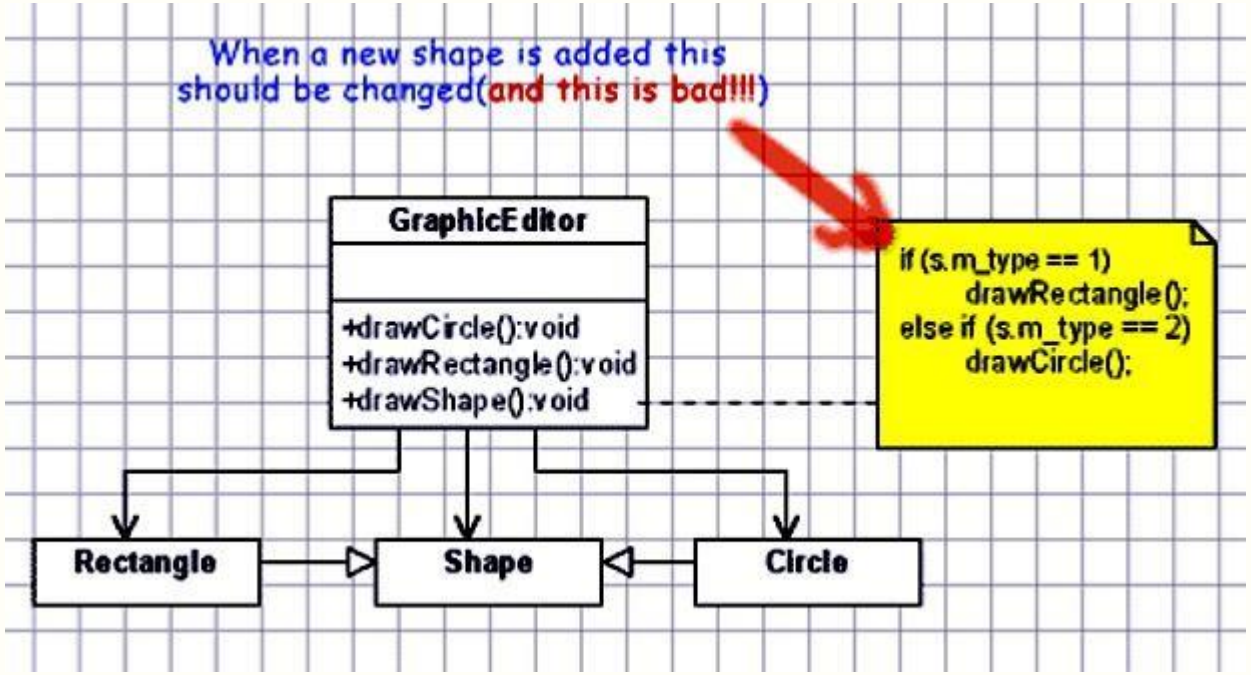



## ❑改后与OCP原则一致

- ➢如果BusinessClient具有对接口的引用，则替换ConcreteServer不会导致BusinessClient的修改



# 例子2： 违法OCP – 开闭原则



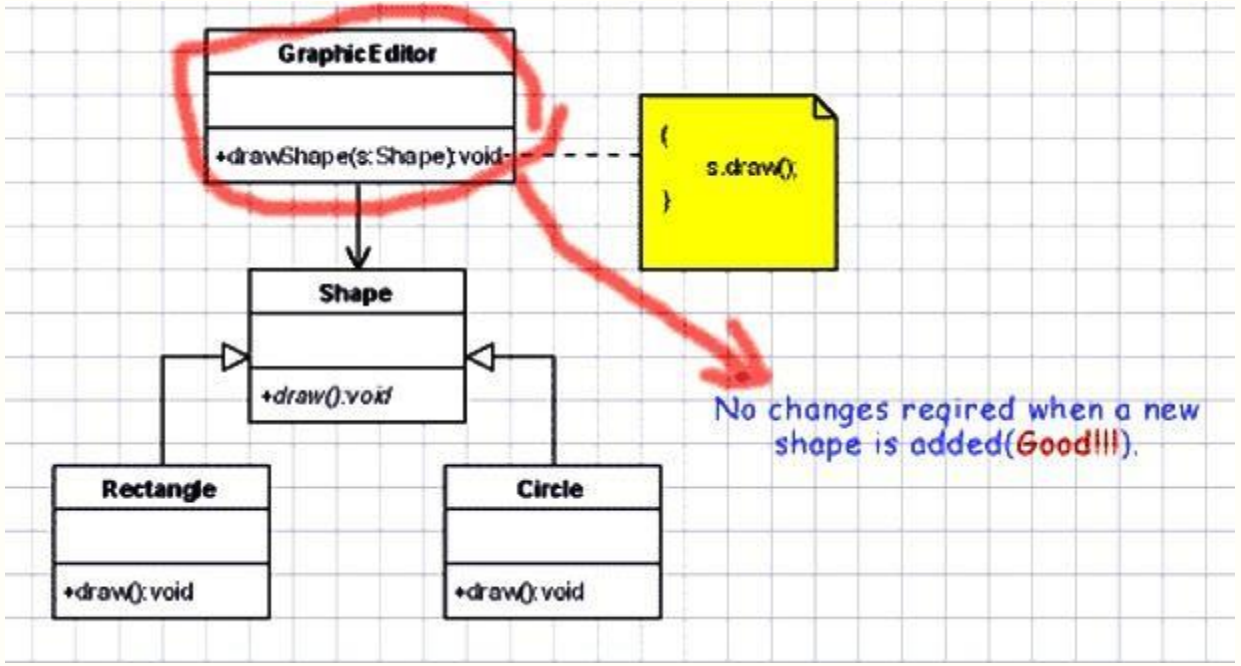

```
// Open-Close Principle - Bad example
class GraphicEditor {
  public void drawShape(Shape s) {
    if (s.m_type==1)
      drawRectangle(s);
    else if (s.m_type==2)
      drawCircle(s);
  }
  public void drawCircle(Circle r) { .... }
  public void drawRectangle(Rectangle r) {
    ....
  }
}
```

```
class Rectangle extends Shape {
  Rectangle() {
    super.m_type=1;
  }
}
```

```
class Shape {
  int m_type;
}
```

```
class Circle extends Shape {
  Circle() {
    super.m_type=2;
  }
}
```

# 例子2:修改使符合OCP – 开闭原则




```
// Open-Close Principle - Good example
class GraphicEditor {
  public void drawShape(Shape s) {
    s.draw();
  }
}
class Shape {
  abstract void draw();
}
class Rectangle extends Shape {
  public void draw() {
    // draw the rectangle
  }

}
```

- 符合 OCP原则，增加一个新的shape:
  - 对扩展开放– 为新的shape增加一个新的子类
  - 对更改封闭– 无需修改 drawShape()

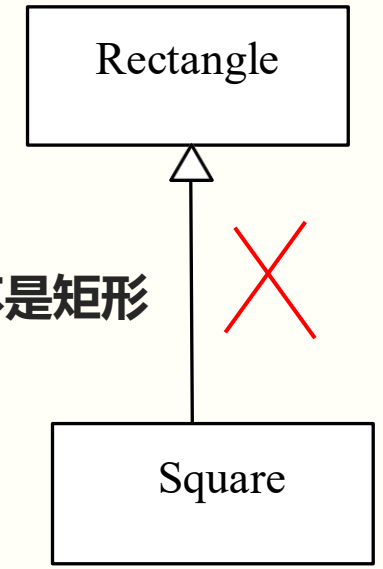
# LSP-里氏替换原则

## ❑LSP: 子类型必须可以替代其基类型

```
class Rectangle {
  int m_width; int m_height;
  void setWidth(int width){ m_width = width; }
  void setHeight(int height){ m_height = height; }
  int getWidth(){ return m_width; }
  int getHeight(){ return m_height; }
  int getArea(){ return m_width * m_height; }
}
```

```
class Square extends Rectangle {
  void setWidth(int width) {
    m_width = width; m_height = width; }
  void setHeight(int height) {
    m_width = height; m_height = height; }
}
```

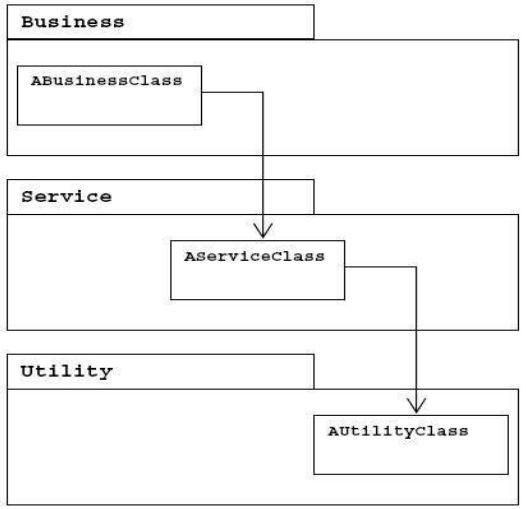
```
class LspTest {
  private static Rectangle getNewRectangle() {
    return new Square();
  }
  public static void main (String args[]) {
    Rectangle r = LspTest.getNewRectangle();
    r.setWidth(5); r.setHeight(10);
    // user knows that r is a rectangle.
    System.out.println(r.getArea());
    // now he's surprised to see that the area is 100 instead of 50.
}}
```




**行为上，正方形不是矩形**

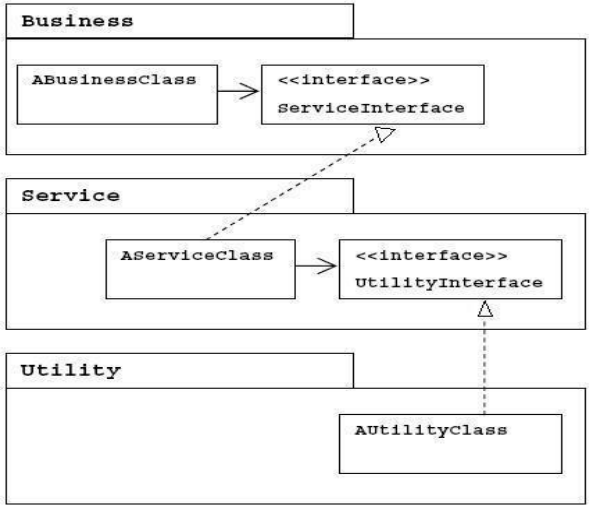
# DIP-依赖倒置原则

- **高层模块不应该依赖低层模块， 两者都应该依赖抽象**
  - 结构化分析与设计往往造成高层模块依赖低层模块，策略依赖细节实现
  - 一般来说，高层模块包含应用程序的商业决策和业务模型
- **抽象不应该依赖细节，细节应该依赖抽象**




**违反DIP原则**

- 高层模块依赖底层模块，低层模块的变化会影响高层模块
- 高层模块很难重用到其他系统中




**与DIP原则一致**

- 采用上层模块发布接口来反转依赖关系
- 业务不再依赖具体的服务，可被重用到其他系统中

# DIP例子I

抽象（抽象类或接口）应该用到系统的易变化部分，以及需要进行解耦的部分

```
// DIP - Bad example
class Worker {
  public void work() {
    // ....working
}}
class Manager {
  Worker m_worker;
  public void setWorker(Worker w) {
    m_worker=w;
  }
  public void manage() {
    m_worker.work();
  }
}
class SuperWorker {
  public void work() {
    //.... working much more
}}
```

高层模块依赖底层模块，更换worker需要修改Manager

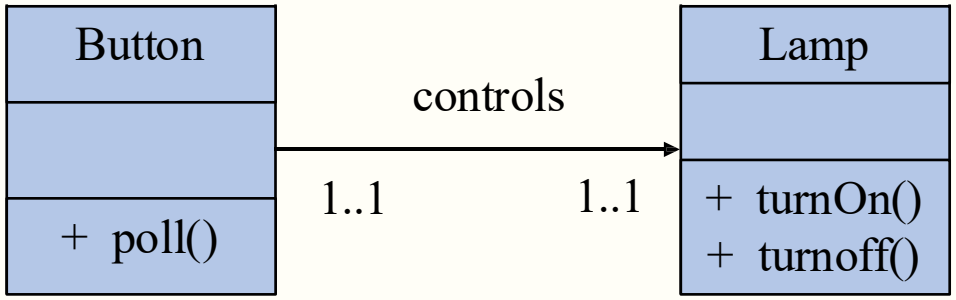
```
// DIP - Good example
interface IWorker { public void work(); }
class Worker implements IWorker{
  public void work() {
    // ....working
}}
class SuperWorker implements IWorker{
  public void work() {
    //.... working much more
}}
class Manager {
  IWorker m_worker;
  public void setWorker(IWorker w) {
    m_worker=w;
  }
  public void manage() {
    m_worker.work();
}}
```

高层模块不依赖底层模块，更换worker不需要修改Manager

# DIP例子II

## ❑违法DIP原则

- ➢按钮依赖灯，不能重用到其他系统中

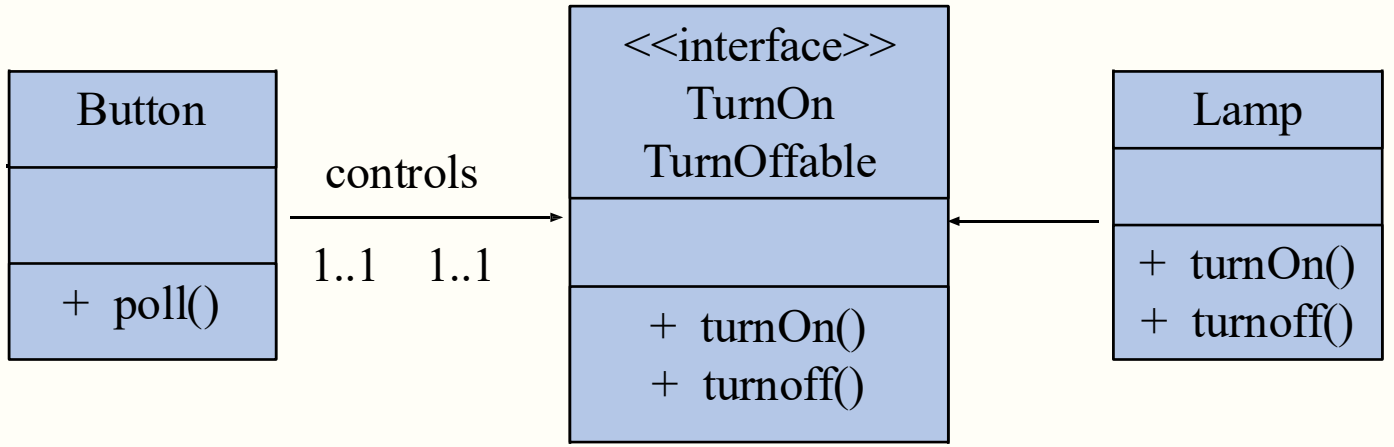



```
Public class Button  {
  private Lamp itsLamp;

  public  void poll()  {
    if (/* some condition */)  itsLamp.turnOn();
  }
}
```

## ❑符合DIP原则

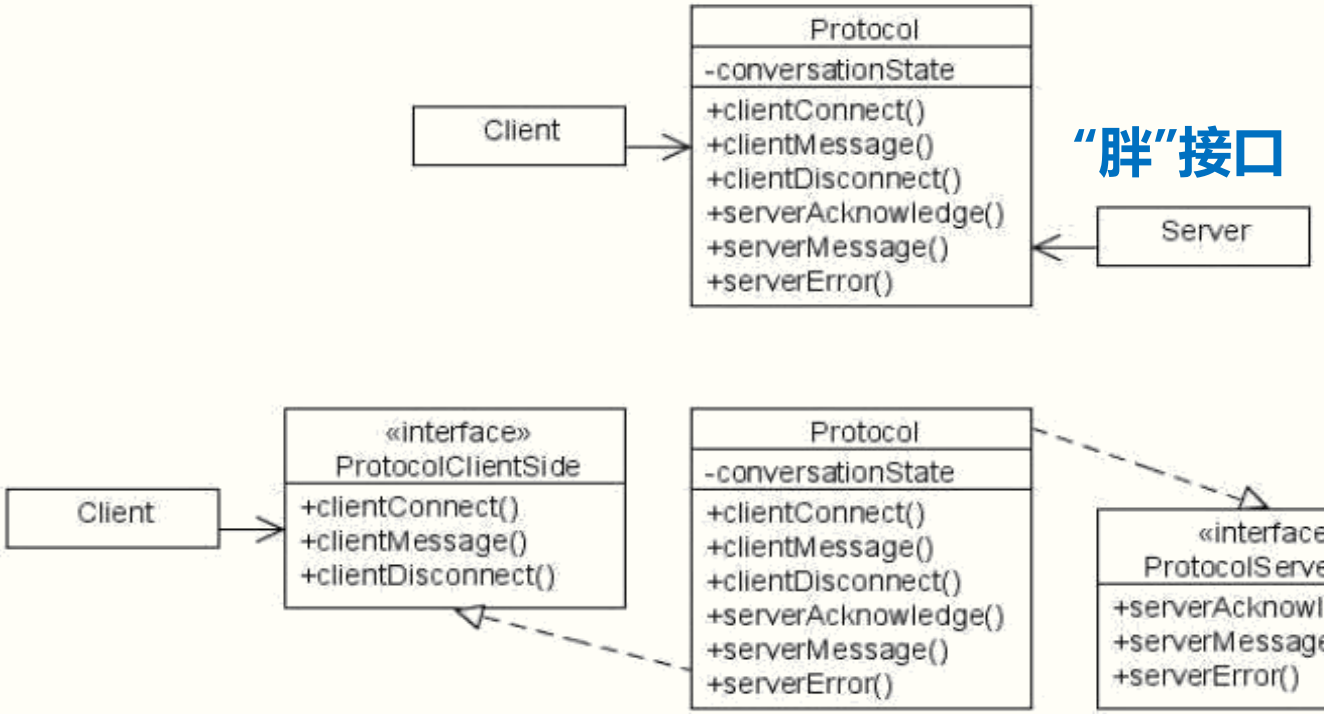
- ➢按钮发布一个接口，使按钮依赖接口；实现该接口的不同电器均可用该按钮控制
- ➢使用此接口的其他控件也可用来控制灯



# ISP-接口分离原则

## ❑客户程序不应该被迫依赖其不使用的方法

- ➢消除多职责的类（接口不内聚的类）
- ➢将“胖”接口分解成多个接口，每个接口服务特定类型的客户程序




- ❑ ”胖“类中多个职责耦合
  - ➢ 一个职责发生变化影响所有clients
- ❑ ”胖“接口应分解为特定于不同clients的多个接口
  - ➢ client不再依赖其不调用的方法，并且clients间不再因”胖“类耦合在一起

# ISP例子

**添加Robot需要实现eat()方法，但其没有该行为**

```
// ISP - bad example
interface IWorker {
  void work(); void eat();
}
class Worker implements IWorker{
  public void work() {
    // ....working
  }
  public void eat() {
    // ...... eating in launch break
}}
class Manager {
  IWorker worker;
  public void setWorker(IWorker w) {
    worker=w;
  }
  public void manage() {
    worker.work();
}}
```

接口分离

```
// ISP - good example
interface IWorker extends Feedable, Workable { }
interface IWorkable {  void work(); }
interface IFeedable{ void eat(); }
class Worker implements IWorkable, IFeedable{
  public void work() { …… }
  public void eat() { …… }
}
class Robot implements IWorkable{
  public void work() { …… }
}
class Manager {
  Workable worker;
  public void setWorker(IWorkable w) {
    worker=w;
  }
  public void manage() {
    worker.work();
}}
```

# 包设计原则

## ❑包的内聚性原则：解决什么样的类放到一个包里的问题

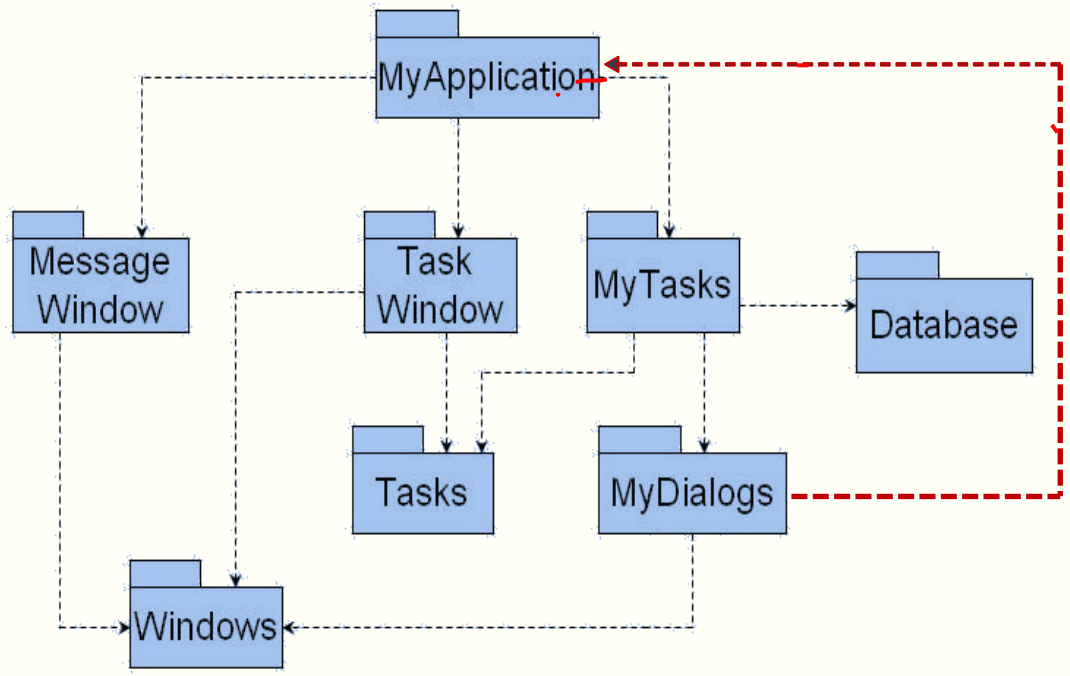
- 重用发布等价原则：复用者的立场来看，软件构件的复用版本和为特殊需求进行修改而生成的发布版本在逻辑上是等价的
- 共同重用原则：包中类彼此依赖，如果重用一个类，则要重用包中所有的类；没有关系的类不应该放在同一个包中
- 共同封闭原则：包中的所有类应该对同一种需求变化封闭；影响包的变化影响包中所有的类，但不影响其他包。

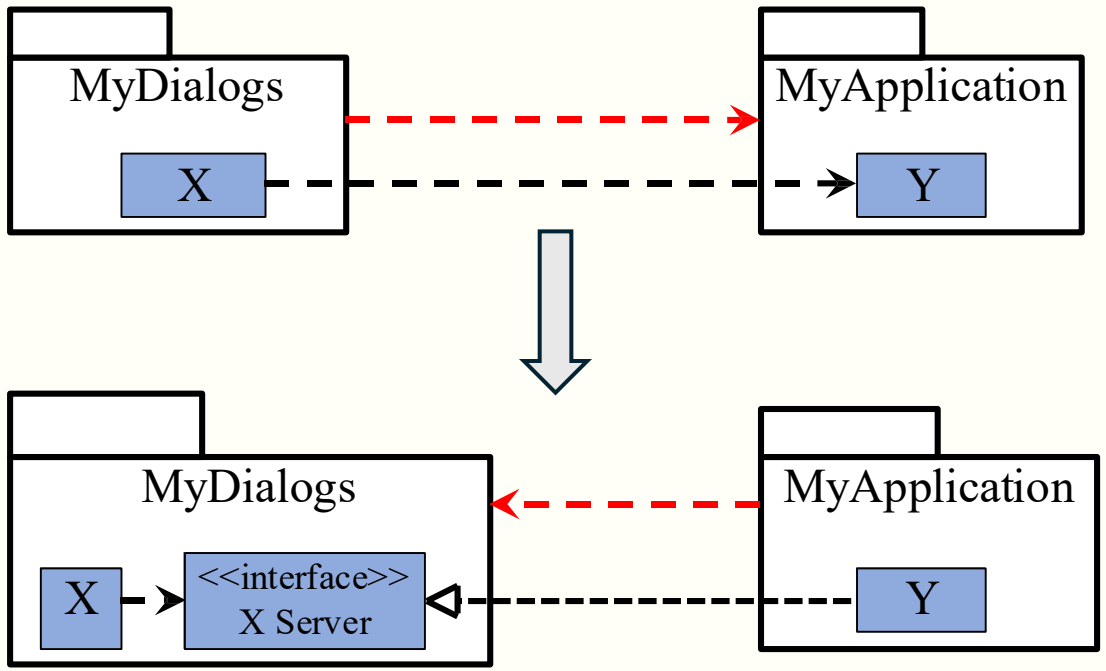
## ❑包的耦合度原则： 解决包之间关系的问题

- 无环依赖原则：在包依赖关系图中不允许环存在
- 稳定依赖原则：顺着稳定的方向依赖，可确保不稳定模块依赖稳定的模块
- 稳定抽象原则：稳定的包应该是抽象的， 其稳定性不应该妨碍它被扩展

# 如何消除包依赖图中的环?

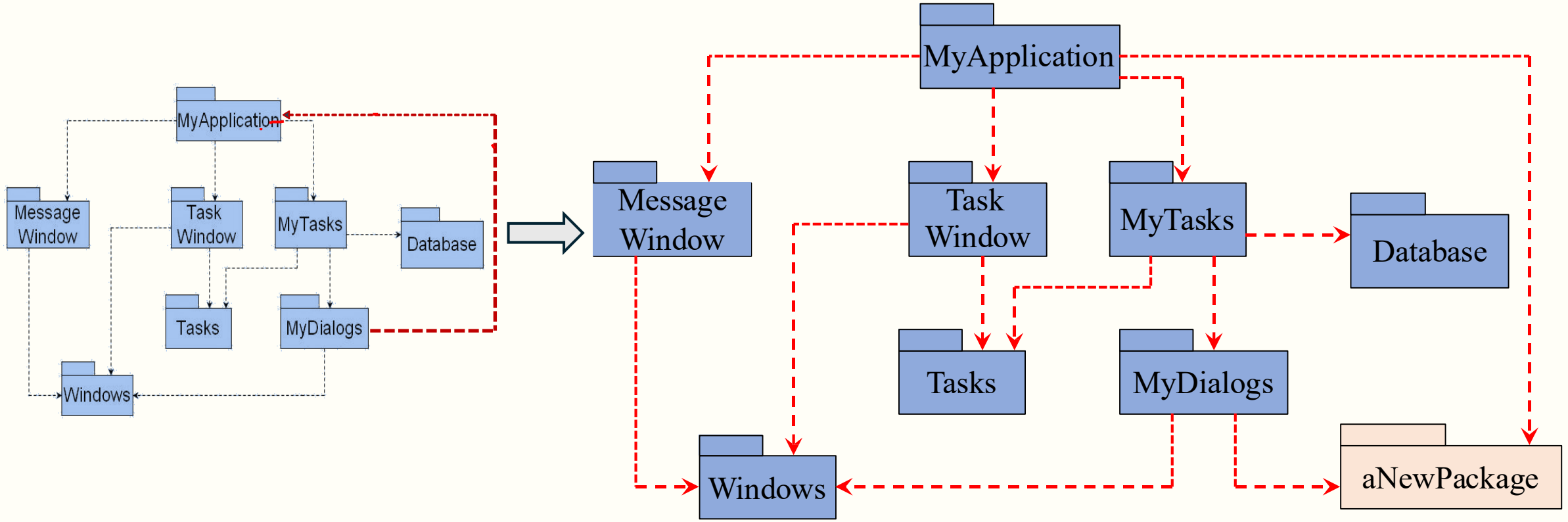
- 用依赖倒置原则解除环



# 如何消除包依赖图中的环?

- 创建新包解除环

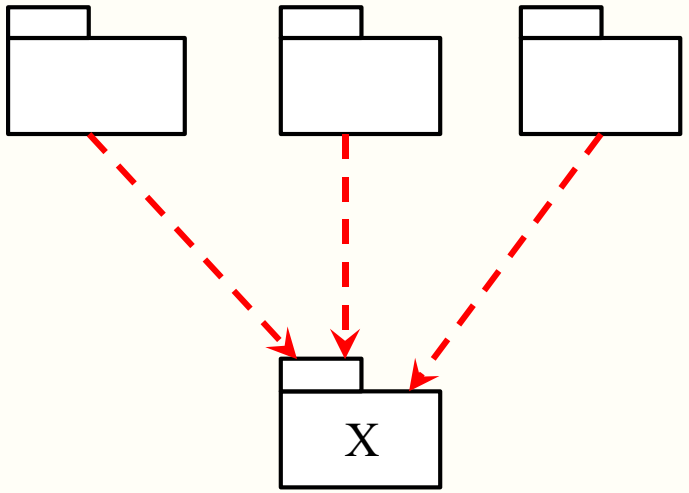


# 如何消除包依赖图中的环？

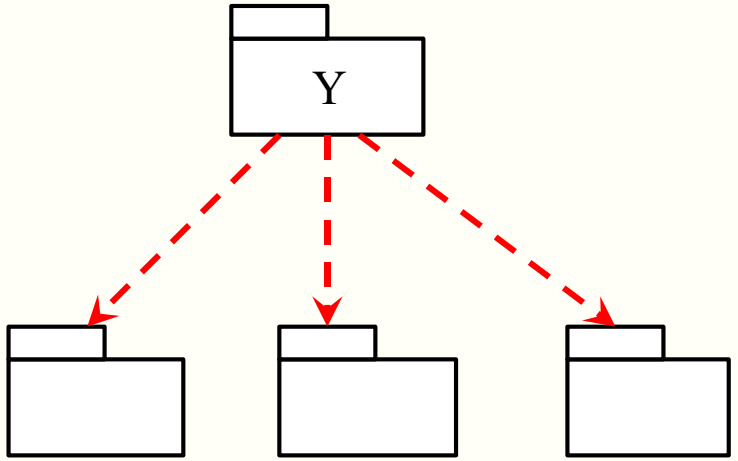
- **采用自底向上增量式开发，不允许环存在**
  - 包的依赖关系图是系统的构建图
  - 包结构不应该从顶到下设计
    - 不应该先设计包
    - 包结构随着系统的增长而演化
  - 包依赖关系无法在类设计之前进行

# 稳定包与不稳定包

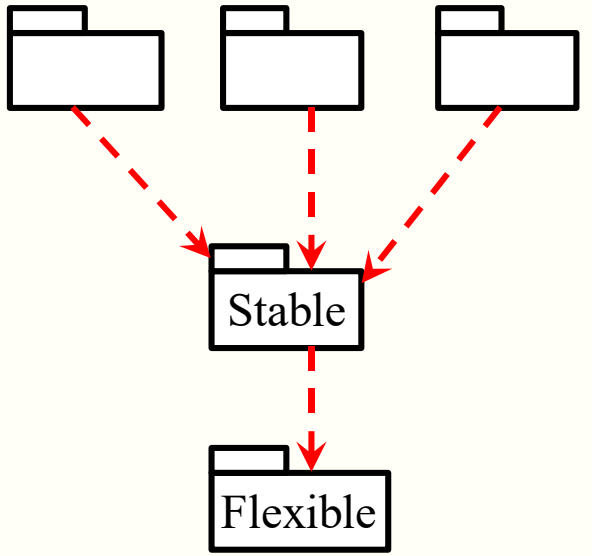
❑X是稳定包：
- ➢被其他包依赖
- ➢但不依赖其他包




❑Y不是稳定包：
- ➢其他其他包
- ➢没有包依赖Y




❑不是所有包都应稳定：
- ➢易变化的包放在顶部
- ➢稳定的包放在下面



# 稳定性和抽象度度量

## ❑稳定性度量：

- ➢$C_a$：包外依赖于包内部类的类的数量
- ➢$C_e$：包内依赖于包外部类的类的数量
- ➢I=0 表示最稳定的包；I=1表示最不稳定的包

$$I = \frac{C_e}{C_a + C_e}$$

## ❑包的$I$值应该大于它依赖的包的$I$值

- ➢即$I$值应该顺着依赖的方向降低

## ❑抽象度度量：

- ➢$N_c$：包中类的数量
- ➢$N_a$：包中抽象类的数量
- ➢A:抽象度

$$A = \frac{Na}{Nc}$$


软件分析与架构设计

# 软件质量属性

**何冬杰**

**重庆大学**

# 什么是质量属性?

- **描述软件系统质量的、精确的、可度量的测度（metric)**
  - 精确描述：它必须能够从最初100个地理上分散的节点扩展到1000个节点，而不会增加安装和配置的工作量和成本
  - 不精确描述： 我的系统一定要是快速的/安全的/可扩展的
- **常见质量属性:**

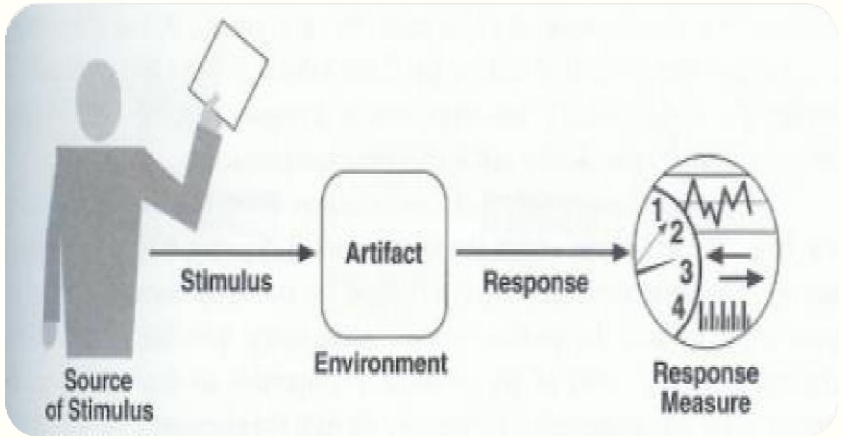
- 性能 (Performance)
- 安全性(Security）
- 可用性（Availability）
- 可扩展性（Scalability)
- 易用性(Usability)
- 可靠性(Reliability)
- 可移植性(Portability)
- 可修改性(Modifiability)
- 可维护性(Maintainability)

# 软件架构与质量属性

- 质量属性的实现必须在设计、实现和部署的过程中考虑
- 从架构方面和非架构方面都要考虑
- 举例：
  - 易用性：
    - 使用户界面易于使用是非架构方面
    - 为用户提供undo/cancel操作是架构方面
  - 可修改性：
    - 功能如何划分(架构方面)
    - 模块内编程技术(非架构方面).
  - 性能：
    - 组件间的通信量&共享资源如何分配 (架构方面)
    - 算法的选择&算法如何实现(非架构方面)

# 质量属性场景

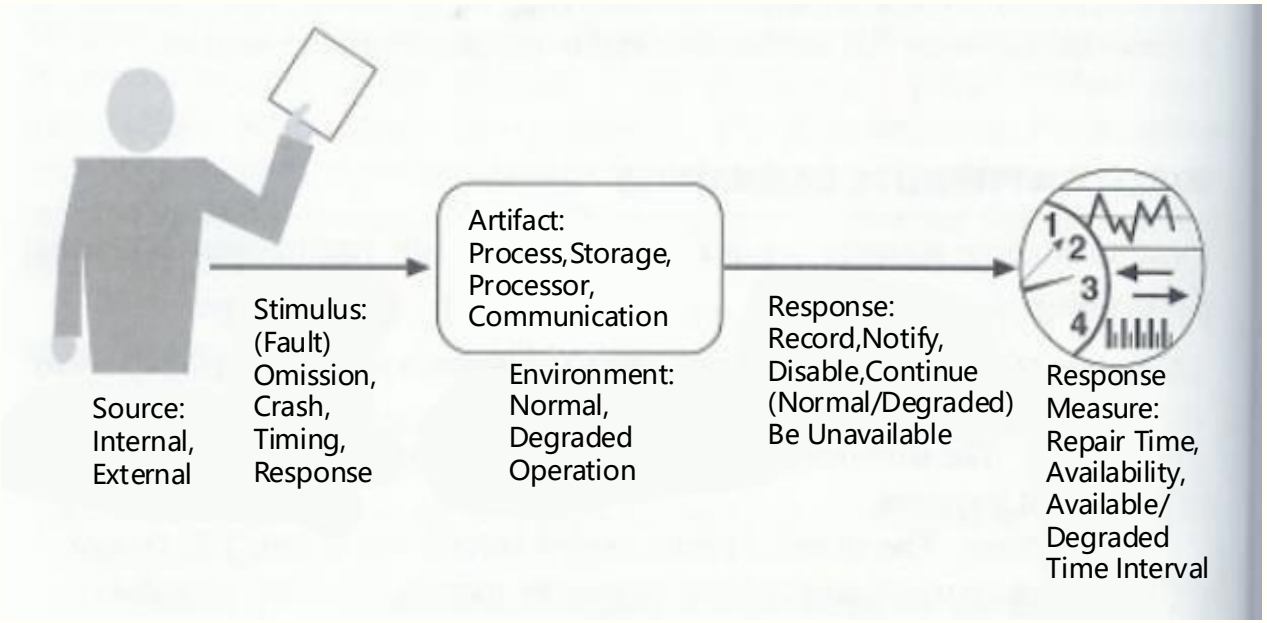
- **一个质量属性特定的需求，由六部分组成：**
  - 激励源– 产生激励的实体
  - 激励 – 影响系统的事件
  - 环境– 激励发生的特定条件
  - Artifact(工件) – 接受激励的系统或系统部分
  - Response(响应) –激励到达后发生的活动
  - Response measure(响应度量)– 响应发生时，应当以某种方式进行度量以测试需求是否被满足



# 通用质量属性场景

## ❑具有系统独立性，适用于任何系统

**可用性的通用场景**

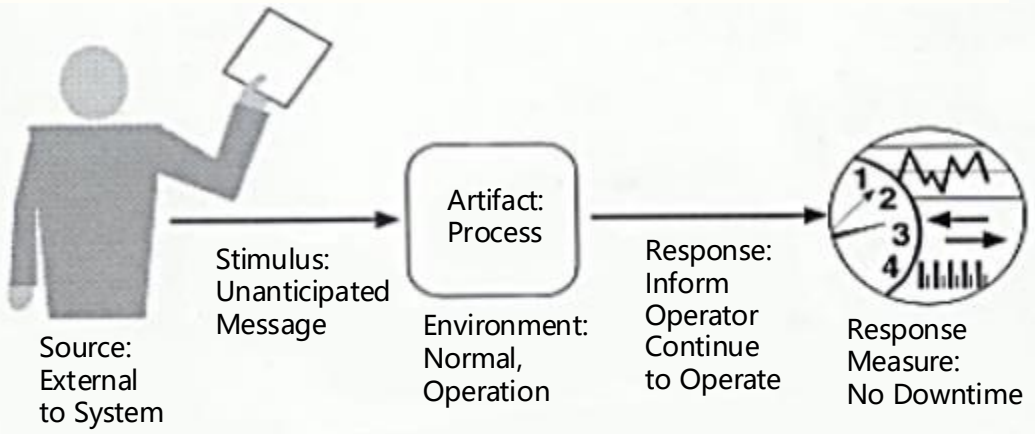


# 具体质量属性场景

## ❑特定于特定系统，让质量需求变得可操作

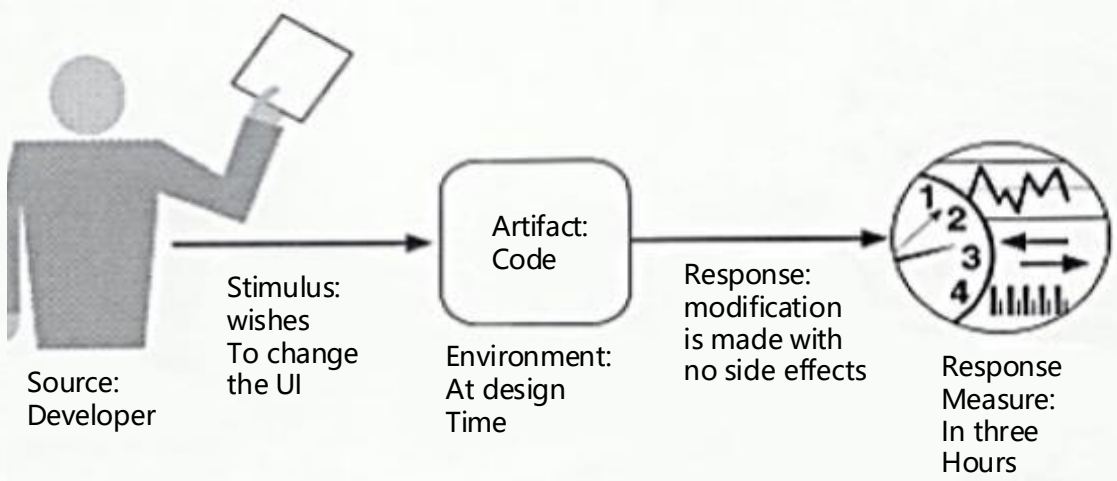
- ➢具体场景集合可用作系统的质量属性需求
- ➢每个场景必须足够具体才对架构师有意义
- ➢响应细节足够具体以便测试系统是否满足了质量属性

### 可用性的具体场景




Sample availabillity scenario. An unanticipated external message is received by a process during normal operation. The process informs the operator of the receipt of the message and the system continues to operate with no downtime.

### 可修改性具体场景




Sample modifiability scenario. A developer wishes to change the UI code at design time; modification is made with no side effects in three hours.

# 质量属性场景生成

- **理论上质量属性需求应在需求分析期间获得，但实际很少能做到**
  - 通过生成具体质量属性场景来确定质量属性需求是架构师的任务

- **特定质量属性表：**
  - 可用来创建通用场景并进而确定具体场景
  - 可充当核对清单，以确保所有可能性都被考虑到，防止重要需求被遗漏
  - 从不同质量属性可能会生成相同或相似的场景，冗余容易被删除

| 场景组成部分 | 可能取值 |
| --- | --- |
| 触发事件 | 系统内部; 系统外部 |
| 触发事件源 | 故障; 遗漏，崩溃，定时，响应 |
| 制品 | 系统的处理器，通信通道，持久存储过程 |
| 环境 | 正常操作模式;降级模式（即更少的功能, 备用解决方案.) |
| 响应 | 系统应当检测事件并能执行以下一项或几项操作：<br>记录它<br>通知相关方，包括用户和其他系统<br>禁用导致错误或失败的事件源<br>在预先设定的时间间隔内不可用<br>继续以正常或降级模式运行 |
| 响应度量 | 系统必须可用的时间区间；可用时间长度； 系统处于降级模式的时间区间；修复时间 |

# 质量属性：性能

## ❑度量系统处理事物的能力

- ➢吞吐量：应用程序在单位时间内必须执行的工作量
  - ○ 每秒处理的事务数量；每分钟处理的消息数
- ➢响应时间： 应用程序处理一个请求的延迟时间
- ➢截止期限：必须在某个特定时间前完成，通常与IT系统中的批处理相关
  - ○ 工资发放系统必须在凌晨两点前结束工作，以便电子转账信息能被发送到银行
  - ○ 每周审计必须在周一早上六点完成以便管理层可以使用数据

## ❑提升性能：

- ➢将关键操作本地化
- ➢尽量减少通信
- ➢使用大粒度的组件

| 场景组成 | 可能的值 |
| --- | --- |
| 激励源 | 多个独立激励源之一，可能来自系统内部 |
| 激励 | 周期性事件;偶然事件； 随机事件 |
| 工件 | 系统 |
| 环境 | 正常模式; 过载模式 |
| 响应 | 处理刺激; 改变服务级别 |
| 响应度量 | 延迟,截止期限, 吞吐量,缺失率, 数据丢失 |

通用性能场景表

# 质量属性：可扩展性

- **描述系统、网络或进程以用户可以接受的方式处理工作量的增长**
  - **请求负载**：从每秒100个请求增加到每秒1000个请求?
  - **连接数**：一个程序的并发连接数增加时会发生什么?
  - **数据大小**：程序或算法能否处理不断增长的数据量?
  - **部署节点数**：当部署节点数增加时，安装、部署的成本如何?

- **可扩展性经常被忽视：**
  - 不是应用崩溃的主要原因
  - 难以预测
  - 难以测试/验证
  - 主要依赖验证过的设计和技术

# 质量属性：可修改性

- **度量修改应用程序以满足新的功能性/非功能性需求的容易程度**
  - 在软件生命周期中对其修改是客观存在的，可修改的系统更容易演化
  - 可修改性只考虑系统可能发生的变化，不需要考虑不太可能发生的变化
  - 可修改性主要考虑变化的成本
- **评估可修改性：**
  - 令人信服的需求变化的影响分析
  - 解决方案在不进行改变的情况下满足需求变化的证明
- **可修改下的影响难以量化，降低依赖性可提高可修改性**

# 质量属性：安全性

- **衡量系统向合法用户提供服务，并阻止未经授权的使用的能力**
  - 破坏安全性的尝试就是攻击
    - 包括非法访问数据、服务或拒绝向合法用户提供服务
- **安全相关概念**
  - **认证：**应用程序可以验证其用户或与之通信的其他应用程序的身份
  - **授权：**被授权的用户和应用程序具有对系统资源的访问权限
  - **加密：**应用程序的消息被加密
  - **完整性：**确保消息内容在传输过程中不被修改
  - **不可否认性：**双方不能否认参与信息交换，交易不能被任意一方否认
  - **审计：** 系统跟踪内部活动并可进行重建

# 质量属性：可用性

- 与系统故障和故障导致的结果相关，通过可用时间的比例来度量
  - 工作时间100%可用
  - 每周计划停机时间不超过两小时
  - 24x7x52 (100%可用)
- 与应用程序的可靠性相关
  - 不可靠的应用程序可用性很差
- 与可用性相关的关注点：

- 系统故障如何被检测到
- 系统故障发生的频率
- 系统故障时会发生什么
- 允许系统停止运行多长时间
- 什么时候故障会安全发生
- 故障如何被避免
- 故障发生时需要什么类型的通知

# 质量属性：易用性

- 易用性涉及用户完成任务的容易程度以及所提供的用户支持类型
- 怎么提升易用性？
  - 学习系统功能
  - 有效使用系统
  - 最小化错误影响
  - 系统适应用户需求
  - 提高用户信心和满意度

| 场景组成部分 | 可能的取值 |
| --- | --- |
| 激励源 | 用户 |
| 激励 | 学习系统功能; 有效使用系统; 最小化错误影响; 适应系统; 感觉舒服 |
| 工件 | 系统 |
| 环境 | 运行时或配置时 |
| 响应 | 系统提供以下一个/多个响应：<br>支持“学习系统功能”——帮助系统对上下文敏感；用户熟悉界面；界面在不熟悉的上下文中使用<br>支持“有效使用系统”——数据/命令的集合; 重用已经输入的数据/命令; 支持屏幕内的高效导航; 具有一致操作的清晰视图; 全面的搜索; 最小化错误的影响：撤消、取消、从系统故障恢复、识别并纠正用户错误、检索忘记的密码，验证系统资源<br>支持“适应系统” – 定制化;国际化<br>支持 “感觉舒服” –显示系统状态; 按照用户的步调工作 |
| 响应度量 | 任务时间，错误数量，解决问题的数量，用户满意度，用户知识的获取，成功操作数占总操作数的比率; 时间/数据丢失量 |

**通用易用性场景表**

# 质量属性冲突与设计权衡

## ❑系统的质量属性经常会互相影响

- ➢影响有时是正面的，有时是负面的
  - ○ 负面：组件粒度大会提高性能，但会降低可维护性
  - ○ 负面：引入冗余数据可提高可用性，但会使安全性的保障更加困难
  - ○ 正面：将安全性相关的功能本地化通常意味着更多的通信以致降低性能

## ❑软件质量属性是系统非功能性需求的一部分

## ❑解决方案要进行合理的折衷：

- ➢不可能完全满足所有相互竞争的需求
- ➢一定要满足所有涉众的需求

软件分析与架构设计

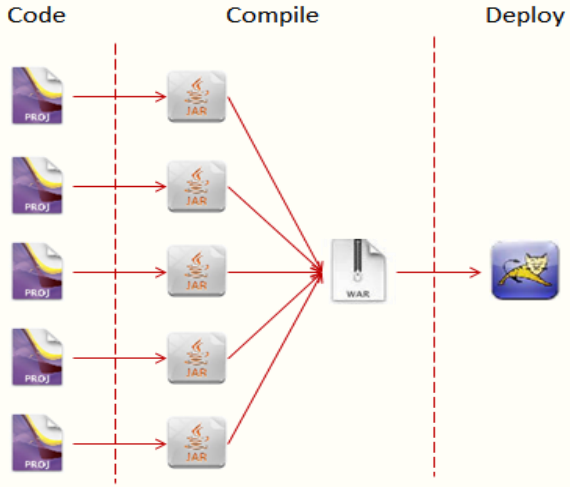
# 软件架构风格

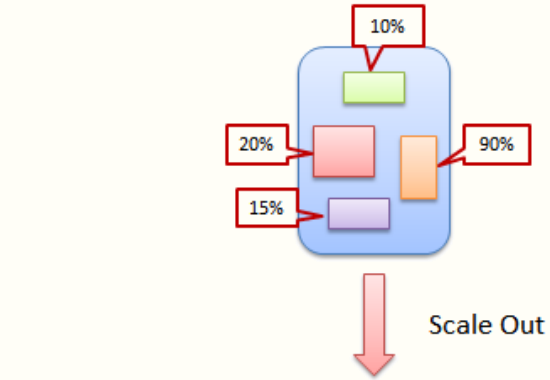
**何冬杰**

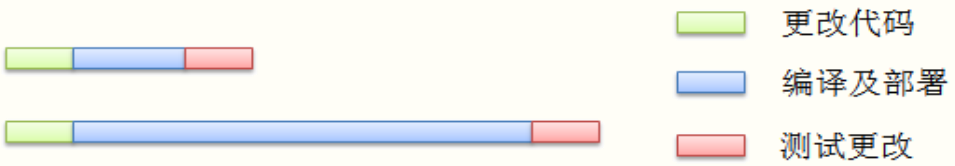
**重庆大学**

# 单体架构

- 软件模块整体打包整体部署
- 缺点：
  - 如果应用部署繁琐，软件开发人员需在部署之前进行大量环境设置，以便测试其更改，使工作变得复杂和枯燥
  - 应用程序扩展时会造成资源浪费
  - 应用程序变大后，软件开发人员花费更多的时间进行编译和部署，超过了其对代码进行更改和测试的时间，效率变得非常低

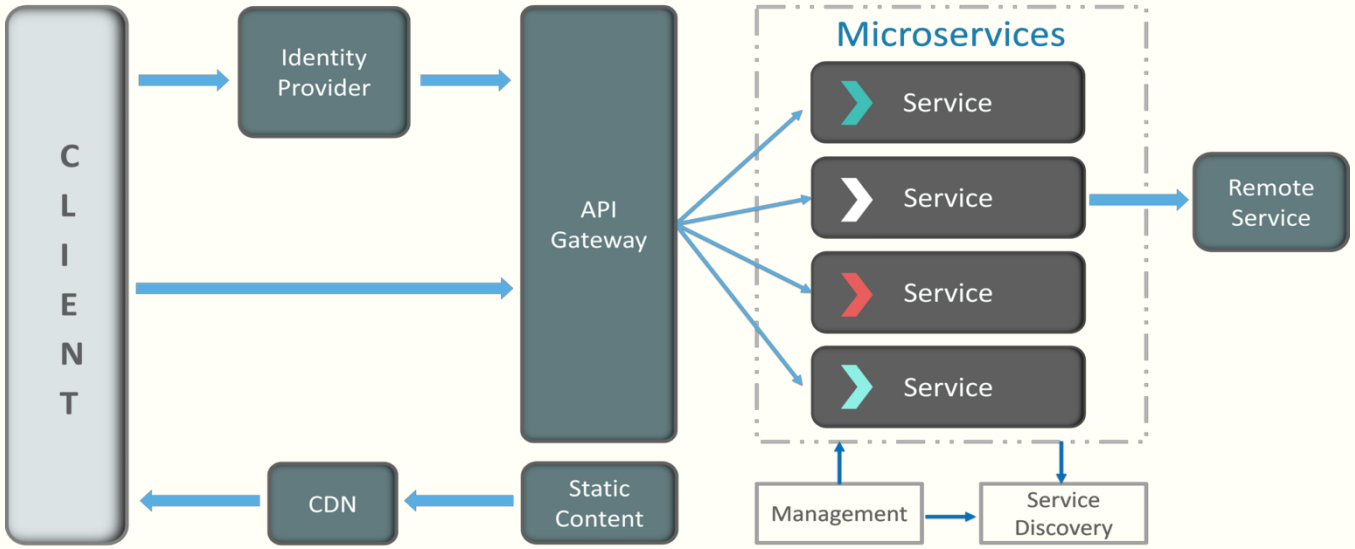


# 微服务架构

- 微服务中大型复杂软件应用程序由一个或多个服务组成
  - 微服务可彼此独立地部署并且耦合松散
  - 每个微服务都专注于完成一项任务
- 架构专注于构建具有良好定义的接口和操作的单一功能模块

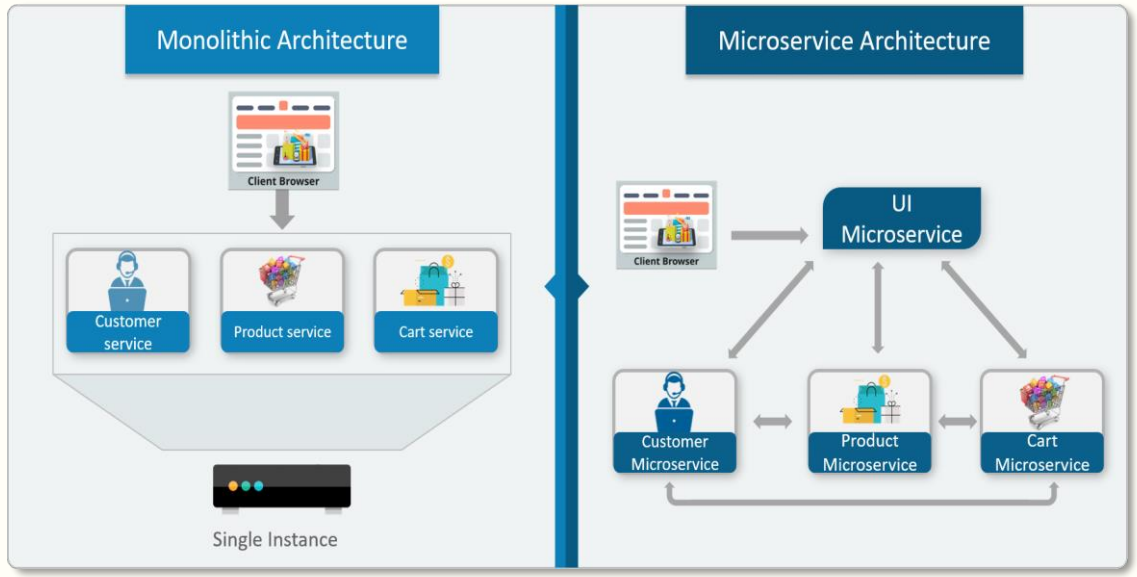


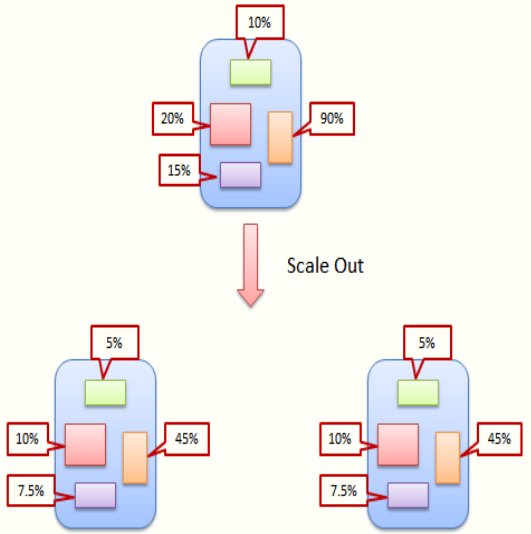

## 微服务架构好处

| | |
|---|---|
| Simpler To Deploy | Deploy in literal pieces without affecting other services. |
| Simpler To Understand | Follow code easier since the function is isolated and less dependent. |
| Reusability Across Business | Share small services like payment or login systems across the business. |
| Faster Defect Isolation | When a test fails or service goes down, isolate it quickly with microservices. |
| Minimized Risk Of Change | Avoid locking in technlogies or languages - change on the fly without risk. |

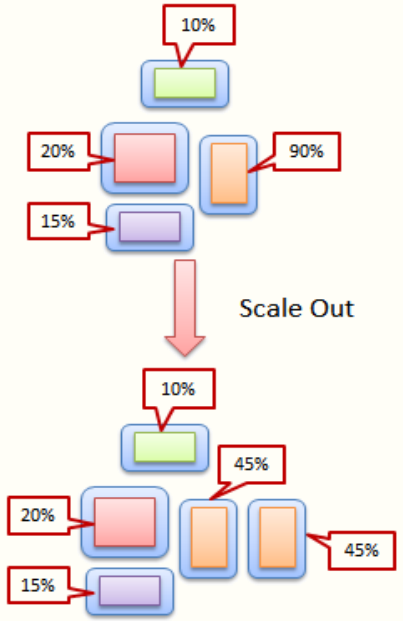
# 单体架构与微服务架构对比

## ❑微服务更容易扩展




单体架构扩展




微服务扩展

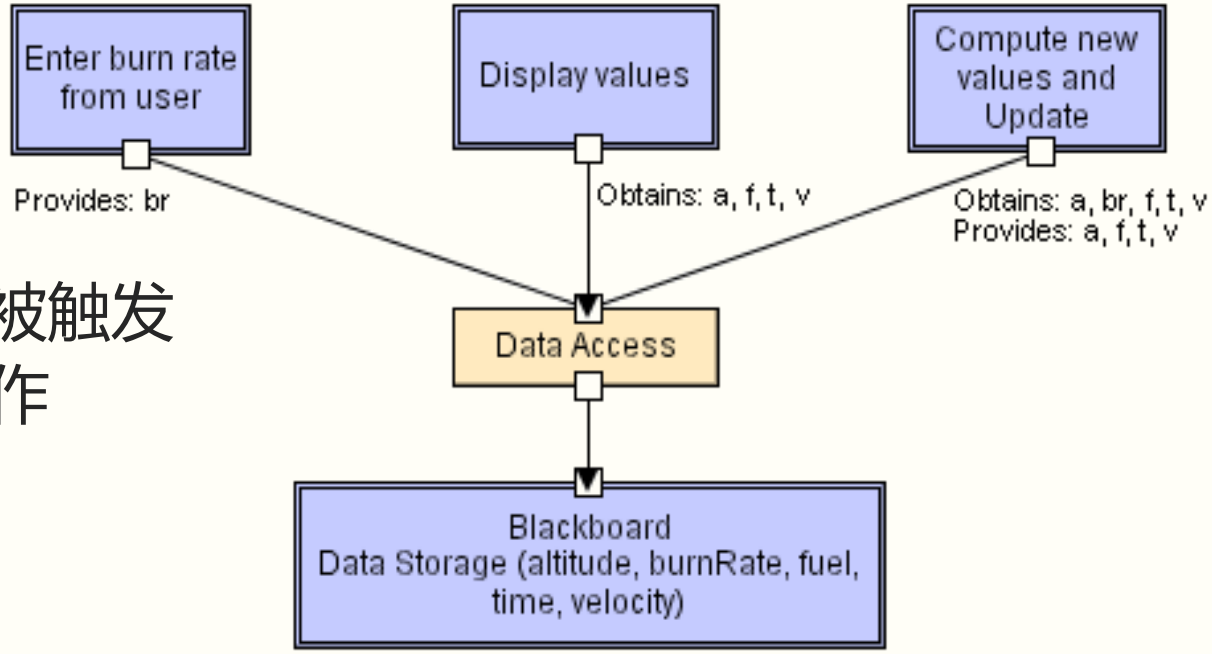
# 仓库模型

- **在需要共享大量数据时使用**
  - 方式1: 共享数据存储在中央数据库或仓库中，能被所有子系统访问
  - 方式2: 每个子系统维护自己的数据库并将数据传送给其他子系统
- **应用领域：**
  - 病人信息管理系统，税务信息管理系统，库存控制系统等
  - 共性：
    - 所有功能都工作于单一数据存储。
    - 数据存储的任何变化都会影响所有或部分功能
    - 所有功能都需要数据存储的信息
- **两种变种：**
  - 黑板风格：数据改变时，数据存储会通知对数据感兴趣的参与方 (触发器)
  - 仓库风格：参与方检查数据存储是否有数据变化

# 仓库模型-黑板风格

- 黑板是中心数据结构，其他组件操作于“黑板”上
- 系统控制完全由“黑板”状态驱动

- 数据库触发器：
  - 事件: 能激活触发器的数据库变化
  - 条件: 护卫条件，为真时，触发器被触发
  - 行动: 当触发器被触发时执行的操作



# 仓库模型的优缺点

## ❑优点：

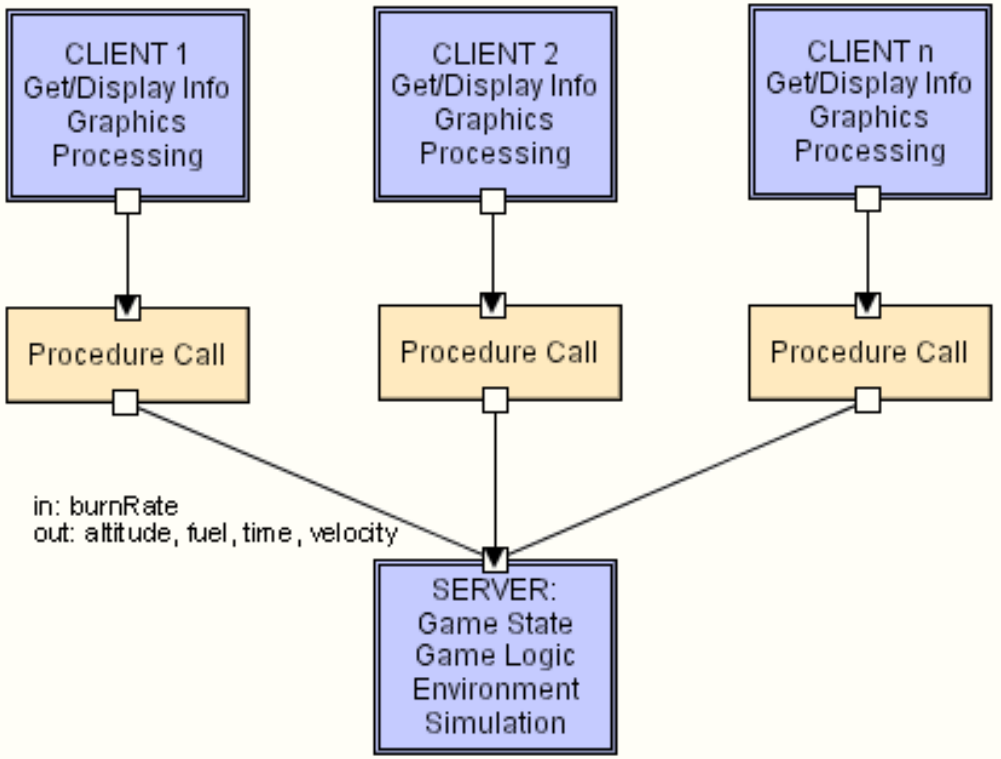
- ➢功能模块本身是内聚的，耦合仅限于共享数据
- ➢单一数据存储使备份恢复和安全性方面的数据维护更容易管理
- ➢能高效共享大规模数据

## ❑缺点：

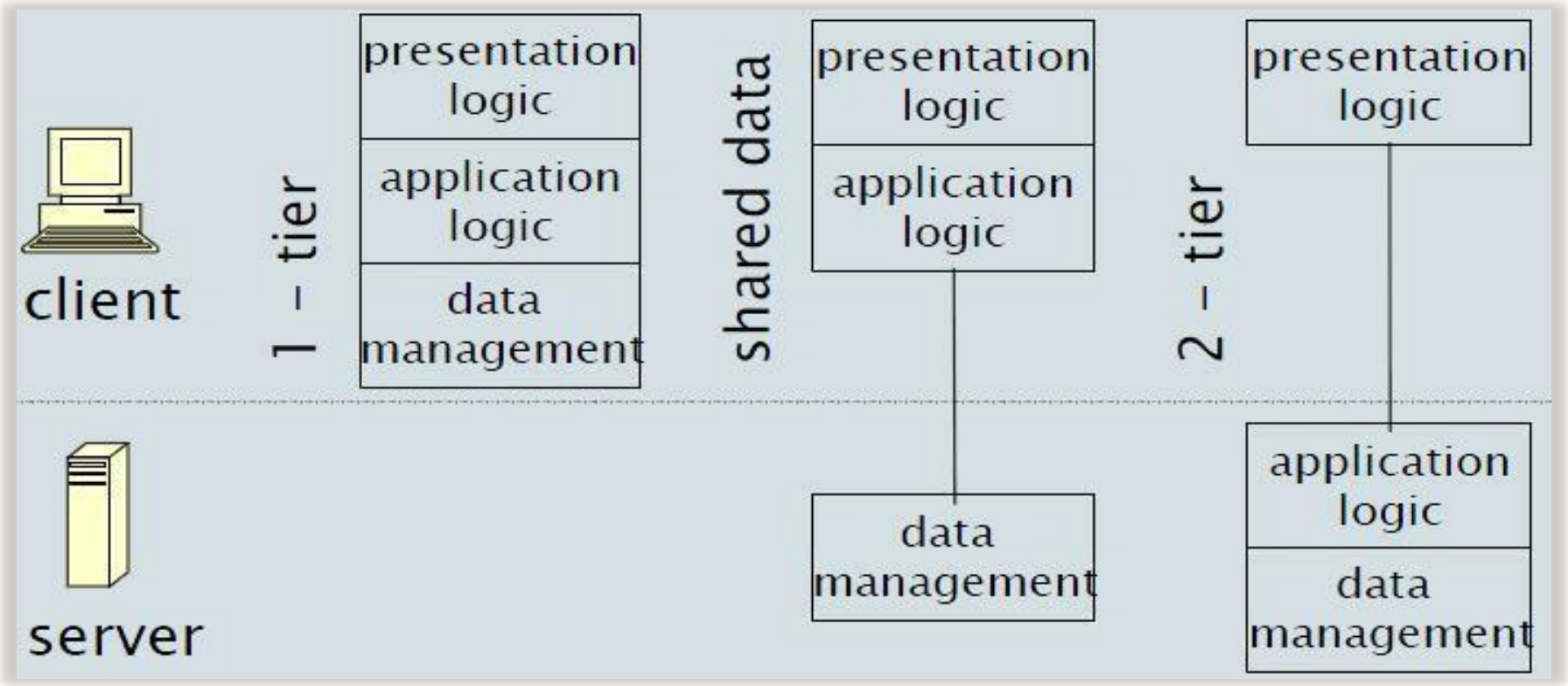
- ➢数据演化困难且昂贵
  - ○共享数据存储中的任何数据格式变化都会影响到全部或部分模块
- ➢难以管理数据
  - ○如果数据存储失败，各方都会受到影响且所有功能都必须停止
  - ○需要冗余数据库和良好的备份和恢复程序

# 客户端-服务器架构

- **客户端：**
  - 对服务器发出请求并处理系统环境中的输入/输出，知道服务器的身份
- **服务器：**
  - 响应客户请求的应用程序，
  - 不知道客户端的数量和身份
- **依赖关系：**客户端依赖服务器
- **拓扑结构**
  - 服务器可能会连接一个或多个客户端
  - 客户端间没有连接
- **移动性：**客户端可移动
- **安全性：**通常在服务器端控制
  - 也可能在应用程序/业务层实现



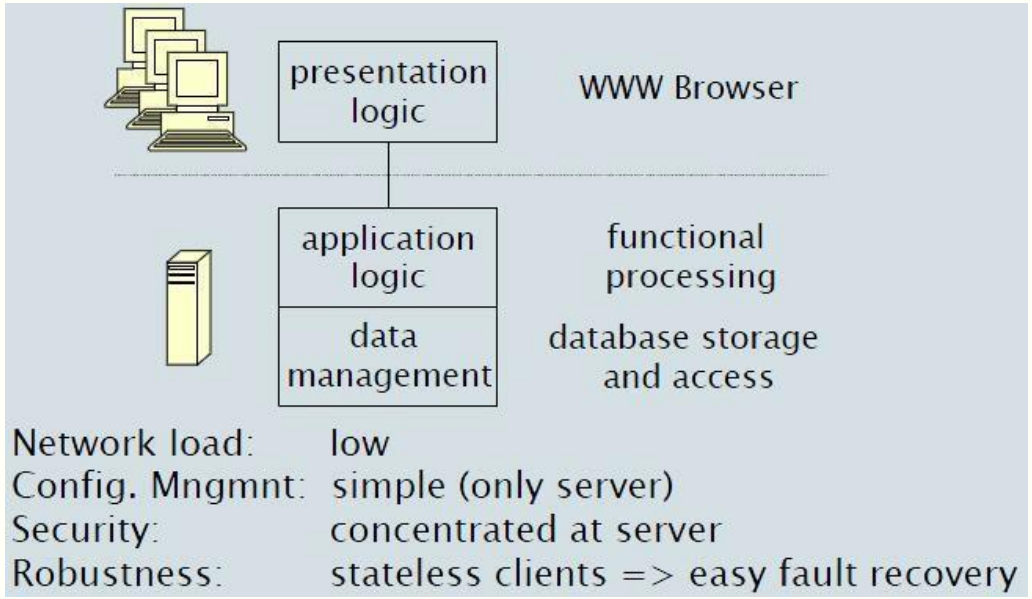
# 一层或两层客户端-服务器架构



# 客户端-服务器架构举例

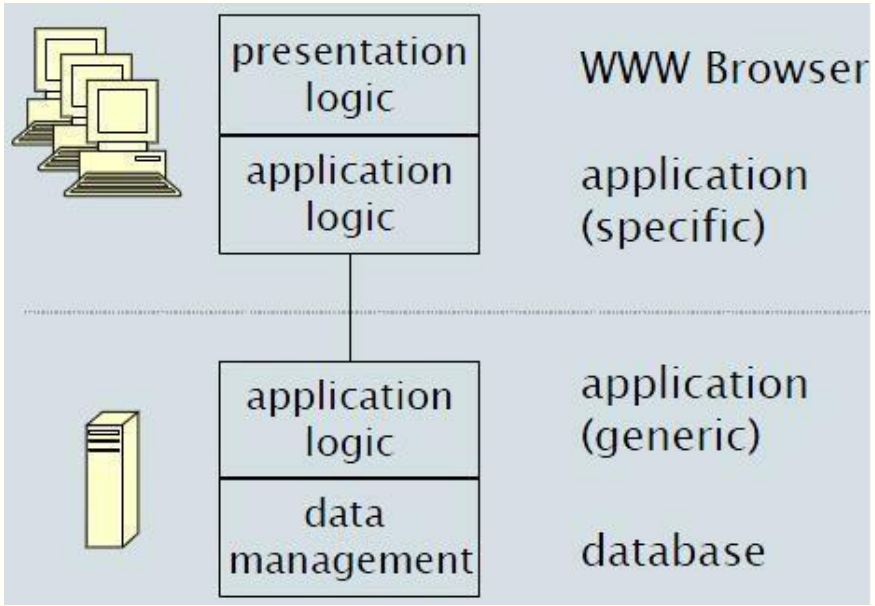
## 瘦客户端

- 服务器端实现大部分业务逻辑处理




## 胖客户端

- 在客户端进行重要的业务逻辑处理



# 客户端-服务器架构特征

## ❑优点

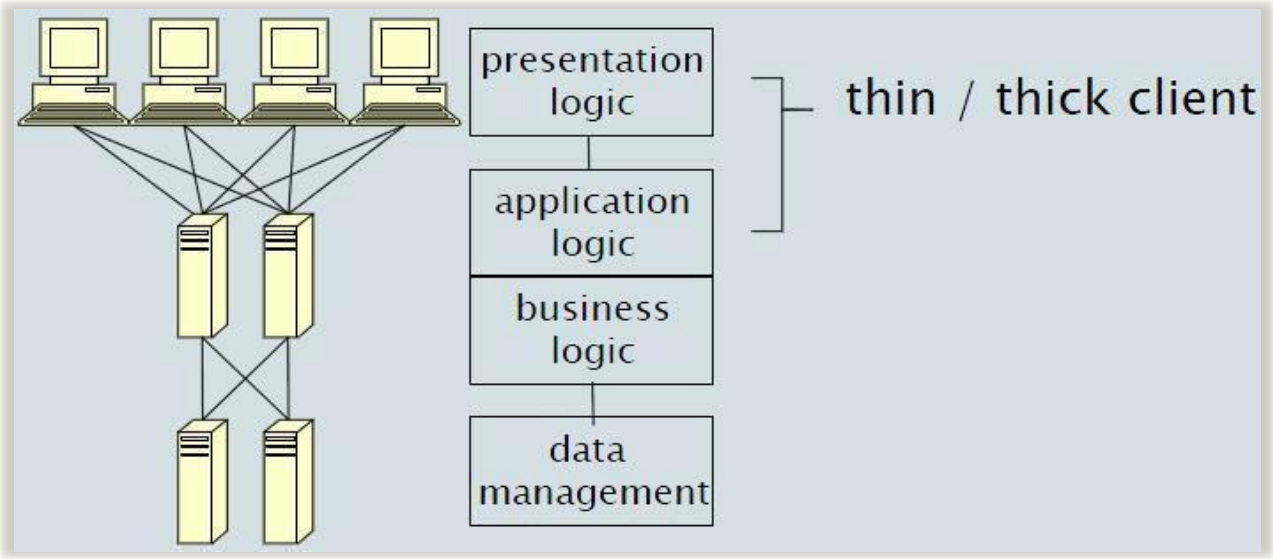
- ➢有效使用网络系统；只需要更便宜的硬件
- ➢易于添加新服务器或升级现有服务器。
- ➢允许多用户间数据共享
- ➢可扩展：添加新的客户端

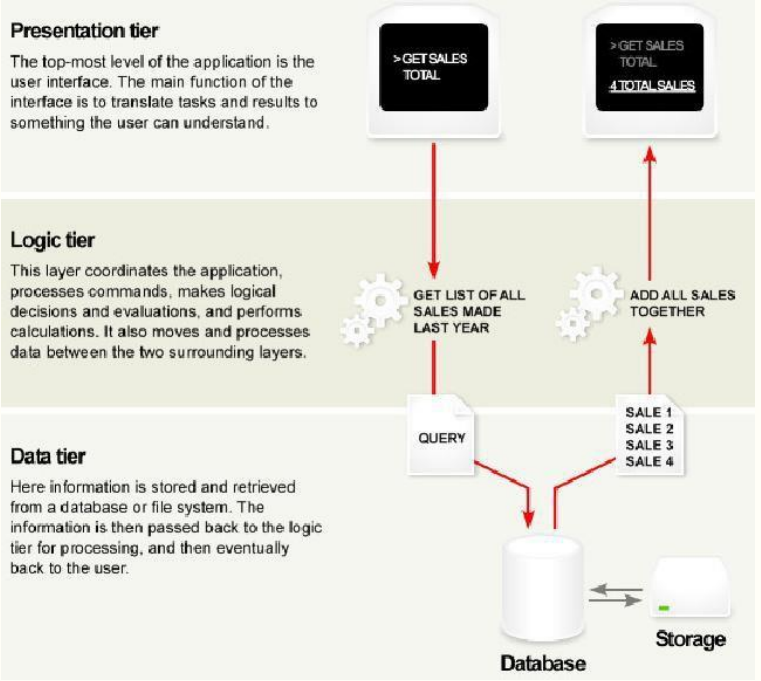
## ❑缺点

- ➢每个服务器上的冗余管理;
- ➢很难发现可用的服务器和服务
- ➢很难改变客户端和服务器的功能
- ➢如果程序逻辑在客户端-服务器上分布，改变程序逻辑很困难
- ➢可扩展性受限于服务器和网络容量的限制

# 三层客户端-服务器架构

- **展示层：**负责显示用户界面的层
- **业务层：**负责访问数据层来获取、修改、删除数据并将结果发送到展示层
- **数据层：** 数据层指的是数据库或者数据源



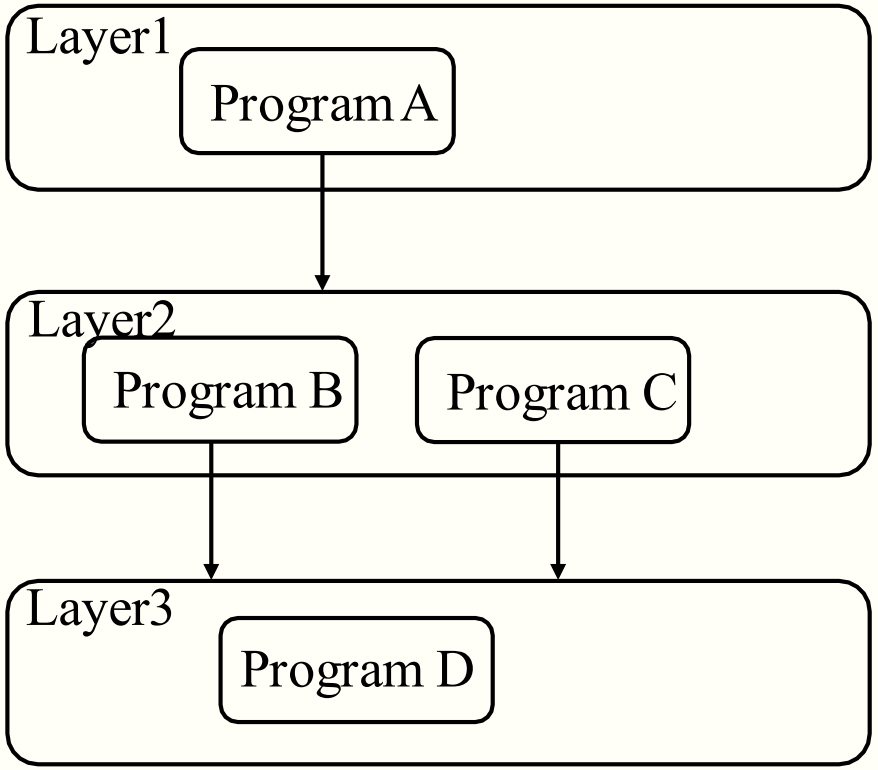



# 三层客户端-服务器架构

- **与两层客户端-服务器架构相比较的优点:**
  - 更好的性能
  - 更好的可扩展性，可重用性和可维护性
  - 与安全性相关的措施可以集中分配
  - 不同层可以平行开发

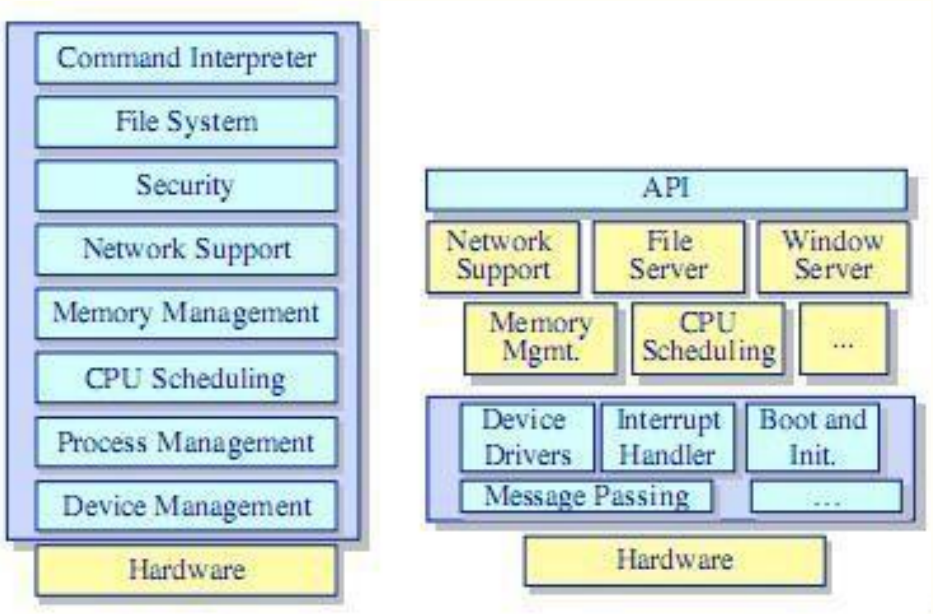
# 分层架构

- **每层提供一组相关的服务;每层只使用下面的层**
  - 如果任意一层只依赖与其直接相邻的下一层，那么它是严格分层结构
  - 如果一层可依赖其下面的任意一层，那么它是宽松分层结构
- **分层系统：“多层客户端-服务器”**
  - 每层暴露一个接口（API）以被上层调用
  - 每层可充当服务器: 给上层提供服务
  - 客户端: 下层服务的消费者

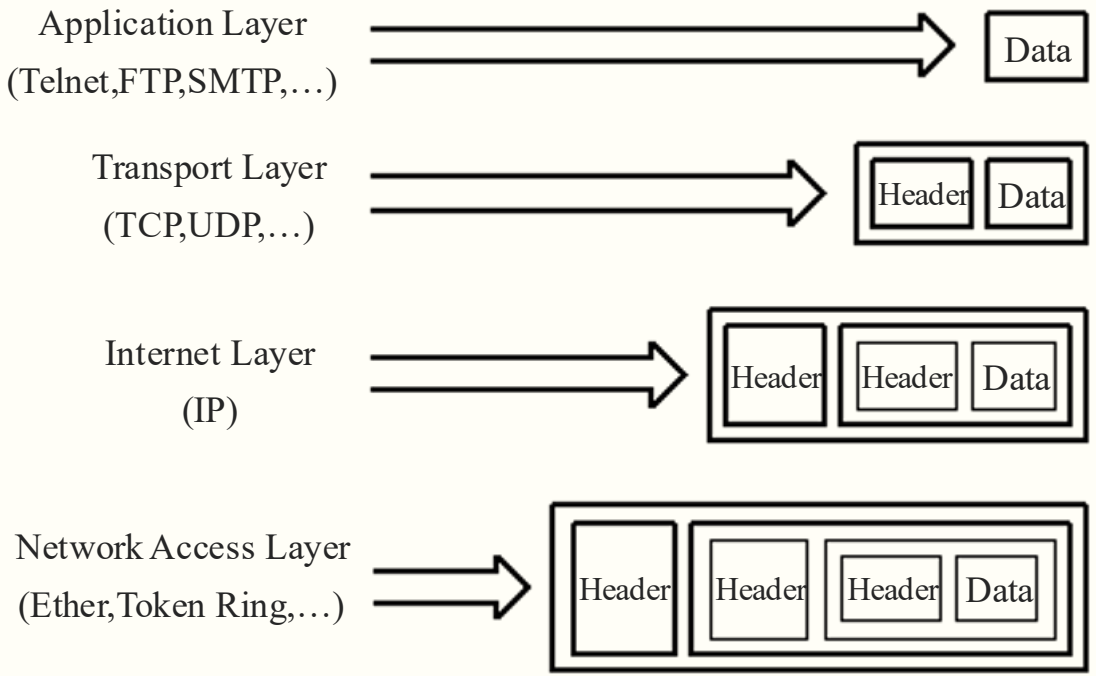


# 常见分层系统

## 单内核OS& 微内核OS




## TCP/IP协议也是分层系统



# 分层系统的优缺点

## ❑优点：层的内聚性高

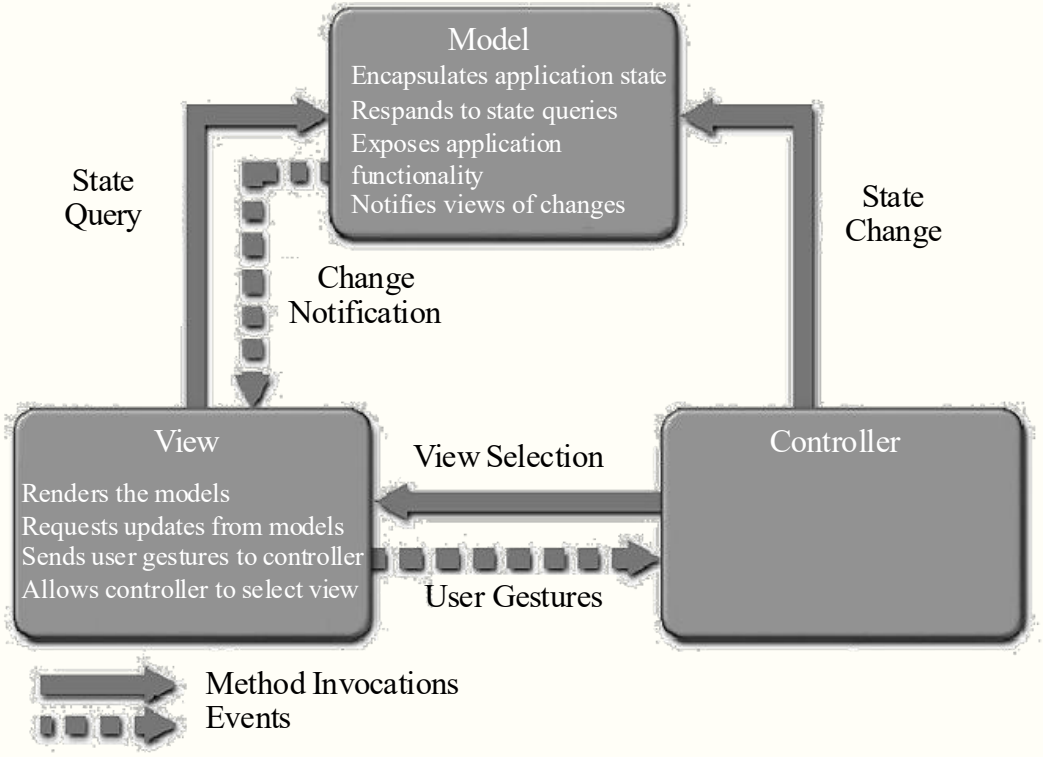
- ➢每层都提供一系列服务，每层都对其他层隐藏私有信息
- ➢每层只用到更低级的层，对某层的修改最多影响相邻的两层
- ➢每层都是内聚的，只和低层耦合因此容易被重用、替换或互换
  - ○数据库的变化只会影响数据存储/访问层，浏览器的变化只会影响展示层，只要接口不变，每层可有不同的实现

## ❑缺点：

- ➢严格的分层可能会导致**性能**问题，具体取决于层数
- ➢建立清晰的分层结构并**不**总是很**容易**

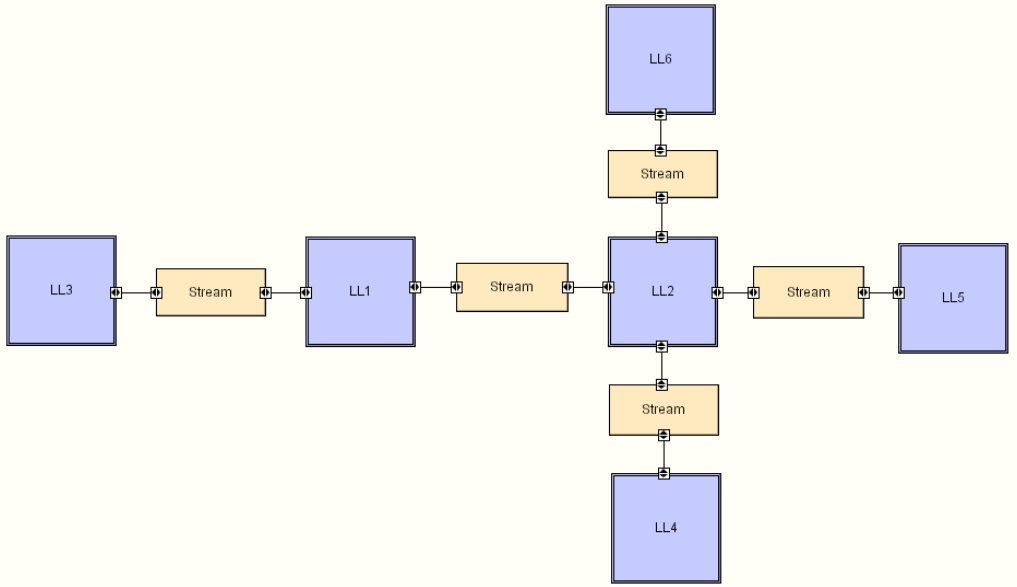
# MVC（模型-视图-控制器）架构

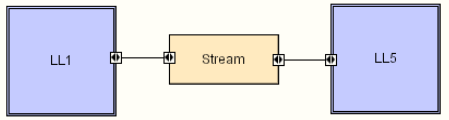
- **优点**
  - 视图，控制器，模型是独立的组件，允许在每层内部进行修改和变化而不用干扰其他层
  - 视图组件经常需要改变（UI技术的改进）和更新来保证用户持续的兴趣。视图组件是独立的。
  - 即使模型层停止工作，视图-控制器也能保持部分功能
- **缺点：**
  - 严重依赖于与MVC架构相匹配的开发和生产系统环境和工具（例如TomCat，.Net，Rail等）



# 点对点（Peer-to-Peer）架构

- 节点之间没有区别
- 每个节点既充当服务器又充当客户端
- 每个节点维护自己的数据存储，以及其他节点地址的动态路由表
- 连接器：
  - 网络协议，通常是自定义的



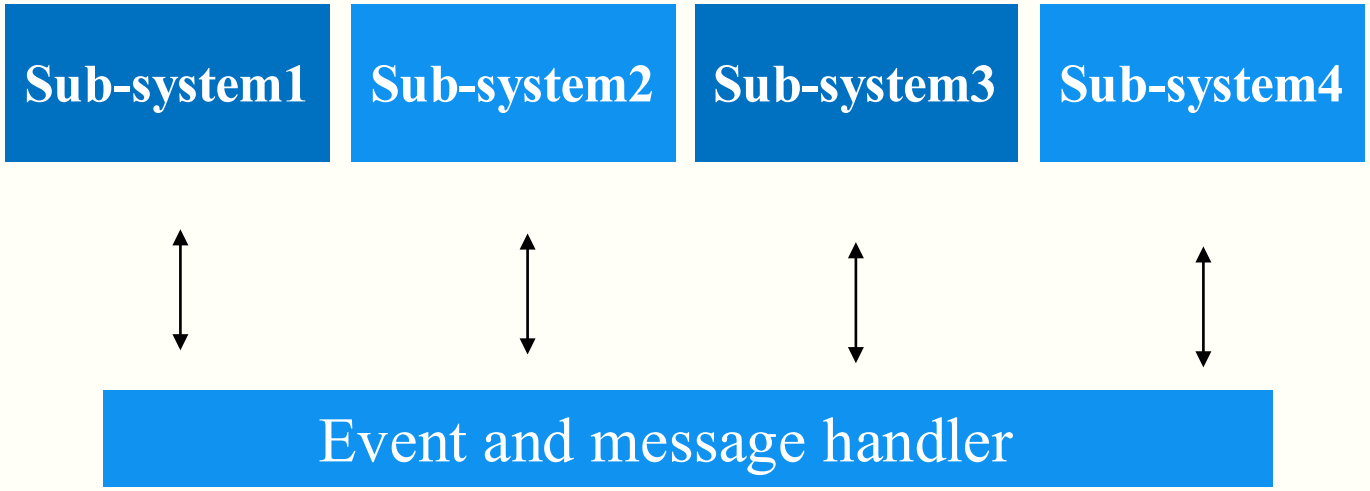
# 事件驱动的（Event-Driven）架构

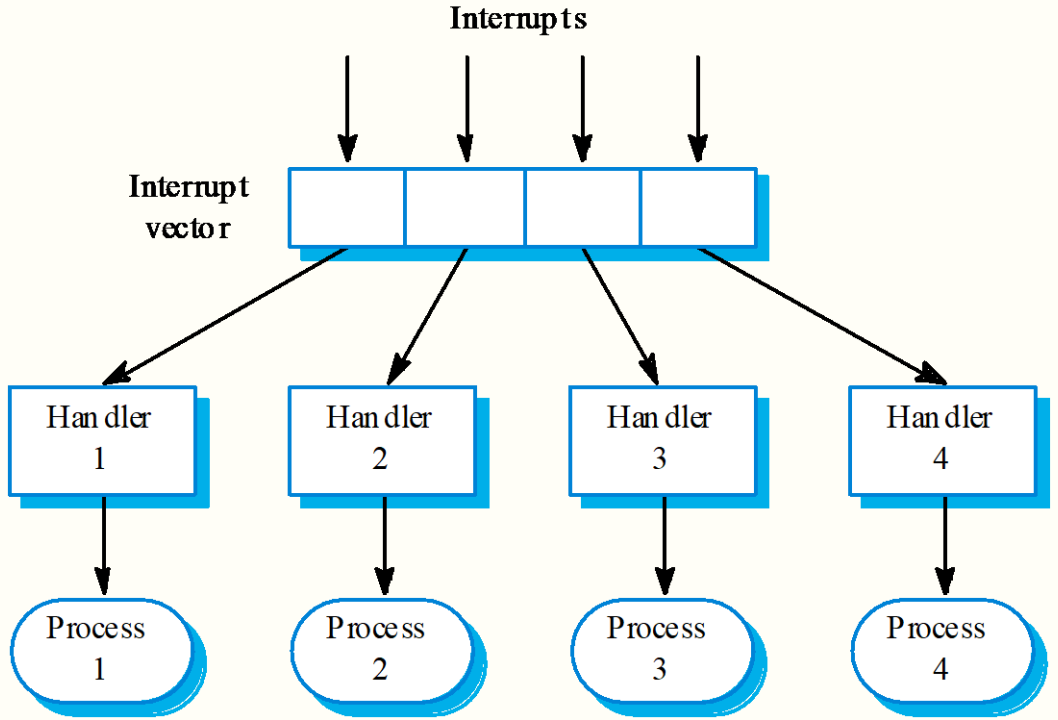
- **基于事件调度程序，调度程序管理事件和基于这些事件的功能**
  - 事件可以是一个简单通知，或可包含关联数据
  - 事件可以基于时间等约束进行优先级设置
  - 事件需要同步或异步处理
  - 事件可由组件“注册”或“取消注册”

- **采用这种架构的系统多为实时系统**
  - 航空控制系统
  - 医疗设备监测系统
  - 家庭监测系统
  - 嵌入式设备控制器
  - 游戏等

# 主要事件驱动的模型

- **广播模型**：一个事件被广播给所有的子系统；任何可处理该事件的子系统接收并进行处理;
- **中断驱动模型**：用于利用中断处理程序检测中断并传递给其他组件以进行处理的实时系统



# 事件驱动架构的优缺点

## ❑优点

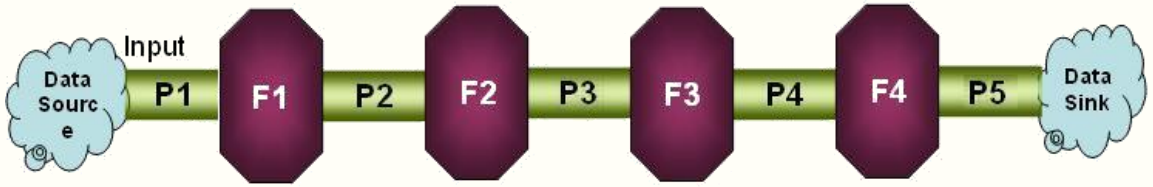
- ➢事件传感器和事件处理器分离（解耦），功能独立
- ➢事件传感器和事件处理器的替换和添加是相互独立的，因此容易执行
- ➢任何传感器或处理器故障都不会影响其他传感器和处理器，可重用性高

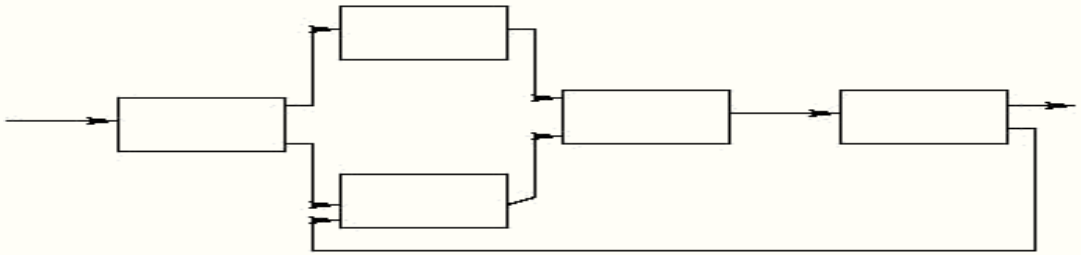
## ❑缺点

- ➢调度器很难对大量的传感器输入作出及时响应（尤其是并发输入）
- ➢调度器故障将导致整个系统停机
- ➢调度器是性能瓶颈，它必须是快速的
- ➢无法保证事件被处理

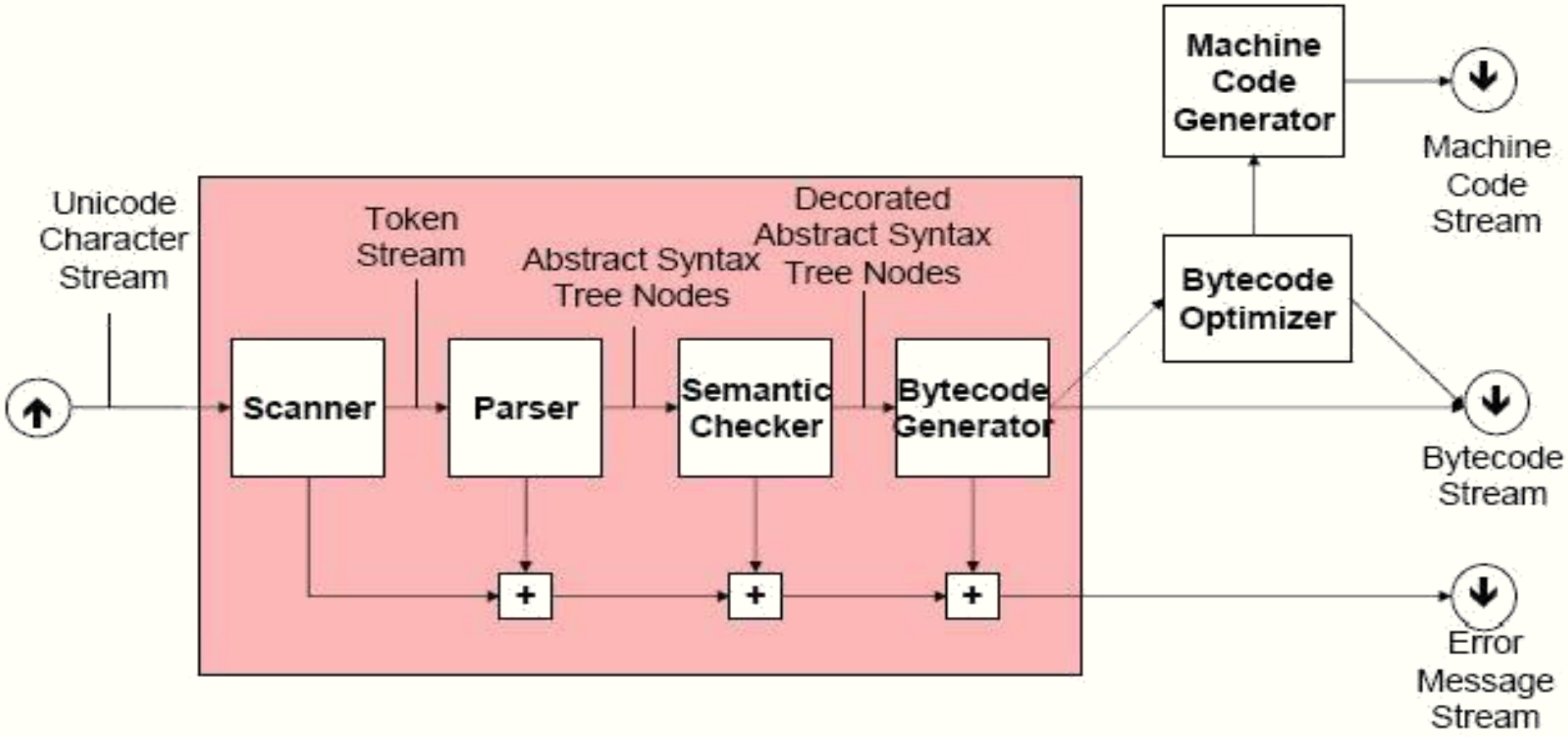
# 管道过滤器模式

- **过滤器 – 处理数据**
  - 从输入读取数据流并输出数据流
  - 过滤器是独立实体，它们不共享状态，它们不知道其前任/后继
- **管道 – 数据翻译与传输**
  - 将过滤器的输出数据传送给其他过滤器
- **管道过滤器模式由一系列处理单元组成，每个单元的输出是下一个单元的输入。在连续的单元之间通常提供一定的缓冲**



# 管道过滤器模式：编译器


Machine Code Generator
Machine Code Stream
Unicode Character Stream
Token Stream
Abstract Syntax Tree Nodes
Decorated Abstract Syntax Tree Nodes
Bytecode Optimizer
Scanner
Parser
Semantic Checker
Bytecode Generator
Bytecode Stream
+
+
+
Error Message Stream

# 管道过滤器模式的优缺点

## ❑优点：

- ➢高内聚：过滤器是执行特定功能的自包含处理服务，具有较强的内聚性
- ➢低耦合: 过滤器间仅通过管道通信
- ➢可重用: 支持过滤器的重用
- ➢可简单地实现为并发或顺序系统
- ➢可扩展性：容易添加新的过滤器
- ➢灵活性：过滤器功能可重新定义，线路可改变

## ❑缺点：

- ➢管道中数据传输需要公共格式
- ➢难以支持基于事件的交互

# （Optional）作业：软件架构分析

- **挑选一个流行的软件系统，分析其架构设计**
  - PDF格式提交
  - 最少1000字的文字说明，语言要求言简意赅，不无病呻吟
  - 可以使用示意图等辅助说明
  - 开放性作业